Introduction to Assembly Language Programming with the Scenix SX Microcontroller

Educational Tutorial for the SX University Program

Version 1.2

PARALLAX INC

Warranty

Parallax warrants its products against defects in materials and workmanship for a period of 90 days. If you discover a defect, Parallax will, at its option, repair, replace, or refund the purchase price. Simply call for a Return Merchandise Authorization (RMA) number, write the number on the outside of the box and send it back to Parallax. Please include your name, telephone number, shipping address, and a description of the problem. We will return your product, or its replacement, using the same shipping method used to ship the product to Parallax.

14-Day Money Back Guarantee

If, within 14 days of having received your product, you find that it does not suit your needs, you may return it for a full refund. Parallax will refund the purchase price of the product, excluding shipping / handling costs. This does not apply if the product has been altered or damaged.

Copyrights and Trademarks

This documentation is copyright 1999 by Parallax, Inc. BASIC Stamp is a registered trademark of Parallax, Inc. If you decided to use the name BASIC Stamp on your web page or in printed material, you must state that "BASIC Stamp is a registered trademark of Parallax, Inc." Other brand and product names are trademarks or registered trademarks of their respective holders.

Disclaimer of Liability

Parallax, Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, and any costs or recovering, reprogramming, or reproducing any data stored in or used with Parallax products. Parallax is also not responsible for any personal damage, including that to life and health, resulting from use of any of our products. You take full responsibility for your BASIC Stamp application, no matter how life-threatening it may be.

Internet Access

We maintain internet systems for your use. These may be used to obtain software, communicate with members of Parallax, and communicate with other customers. Access information is shown below:

E-mail: sxtech@parallaxinc.com
Ftp: ftp.parallaxinc.com - ftp.stampsinclass.com - ftp.sxtech.com
Web: http://www.parallaxinc.com - http://www.stampsinclass.com - http://www.sxtech.com

Table of Contents

Unit I. Getting Started **7**
About This Course 7
Start at the Beginning 8
Problem #1 9
Problem #2 10
Watch Your Language 10
The Working Environment 11
Is That It? 11
The Development Cycle 12
Number Systems 13
Other Places, Other Bases 14
Say What You Mean 15
Size Matters 15
The Hardware Connection 15
Summary 16
Exercises 16
Answers 17

Unit II. Your First Program **19**
First Step 19
Lock and Load 20
So What? 21
Inside the Program 21
Registers 22
Elementary Debugging 23
Stopping the Debugger 26
Summary 27
Exercises 27
Answers 28

Unit III. Simple Flow Control **29**
Running? 29
More Interesting? 30
What's Wrong? 31
Other Forms of JMP 32
Local Labels 33
Another Way to INC 33
Stopping the Processor 34
About the Watchdog 34
Summary 36
Exercises 37
Answers 38

Unit IV. Variables and Math **39**
An Example 40
Assignment 42
Performing Math 43
Two's Compliment Numbers 44
More Carry Tricks 45
Try It! 45
A Few More Functions 46
Programmed Delays 48
Logical Functions 49
Summary 52
Exercises 52
Answers 53

Unit V. Advanced Flow Control **55**
Comparing 56
Using Call and Return 57
Tables 61
Math Functions 63
Division 64
Summary 66
Exercises 66
Answers 68

Unit VI. Low-Level Programming **73**
Port Control 73
Analog Capabilities 75
Register Banking 75
Program Pages 78
Reading Program Storage 79
Summary 79
Exercises 80
Answers 81

Unit VII. Interrupts **83**
The Real-Time Clock Counter 83
RTCC Delays 85
RTCC Interrupts 85
Periodic Interrupts 87
A Clock Example 88
External Interrupts via RTCC 90
Port B Multi-Input Wakeup 90
Port B Interrupts 93

Summary....94
Exercises....94
Answers....95

Unit VIII. Virtual Peripherals....101
Using a Virtual Peripheral....101
Mixing Virtual Peripherals....103
Summary....104
Exercises....104
Answers....105

Appendix A. Instruction Summary....115
Appendix B. Hardware....123

Unit I. Getting Started

Unit I from Introduction to Assembly Language with the Scenix SX Microcontroller

© 1999 by Parallax, Inc. All Rights Reserved. By Al Williams, AWC

Back in 1943, the chairman of IBM predicted that one day there would be a world-wide market for five computers. Today, computers are everywhere. Sure, there are PCs in many homes, but the real computer invasion isn't in the home PC. Instead, people buy computers in just about every electronic device they own. Today your television, your phone, your microwave oven, and your car all have computers (some have several computers).

These computers may not be as obviously powerful as your desktop PC, but they are designed to control the real-world. An integral part to designing electronic equipment today (for fun or for profit) is understanding how these devices work and how you can use them in your own creations.

Why use these microcontrollers? Often a microcontroller can replace a large number of other components. For example, consider a phone answering machine. Do you really need a microcontroller to do the job? No. If fact, many old fashioned answering machines did not use microcontrollers. Instead they had a circuit to detect a ringing phone. The ringing would activate a timer chip (or in a really old machine a timing cam on a motor). This timer would trip a relay that would take the phone off the hook. Then another timer would start the tape player that played the outgoing message. When the outgoing message finished (based on time, or sensing a clear piece of tape at the end of the tape), another timer would start a regular tape recorder for a preset time to record the call.

Instead of three timers today's answering machine uses a microcontroller. With just a few external parts, the microcontroller can operate the entire system with ease. But there is much more. A microcontroller can also sense if someone is really talking on the other end of the line. It can accept Touch-Tone commands to allow remote control. It can even store and playback voice digitally instead of using tapes. Try making a sophisticated remote control without a microcontroller.

So our microcontroller phone machine is much more powerful than its ancestors. It also costs less. Microcontrollers are now quite inexpensive - even if you don't account for the number of parts it can replace. Fewer parts also make devices smaller, cheaper, and less prone to failure.

About This Course

This course is all about incorporating these powerful little computers - microcontrollers - into your own designs. Particularly, we will use the Scenix SX microcontroller along with the SX-Key development system from Parallax. The SX is an inexpensive yet very powerful microcontroller. The SX-Key allows you to program the SX and also debug your programs in real-time. In the past, hardware like the SX-Key was very expensive (thousands of dollars) and was only available to well-stocked labs. However, the SX-Key is quite affordable (only a few hundred dollars, depending on options).

To get the most out of this course, you should already be familiar with elementary hardware design. You should understand how LEDs work, for example, and understand basic electronic laws (like Ohm's law). This course will focus on designing programs to run the microcontroller and thereby control electronic circuits. Although you usually think of programs as software, when a program is inside a microcontroller it is often known as firmware - a cross between hardware and software.

The labs in this course are best performed with the SX-Tech Board available from Parallax. However, you can also wire up your own version of these circuits on any solderless breadboard (See Appendix B).

The SX chip is a very powerful chip, but is also useful as a learning tool. Unlike some microcontrollers, the SX uses electrically erasable memory to store programs. That means that you can write a program, try it, and then reprogram the chip immediately to run a different program (or a corrected version of the same program). This coupled with the powerful SX-Key tools provides an ideal environment for learning and experimentation.

Start at the Beginning

If you are not familiar with the way a computer operates internally, it can seem like black magic. It seems as though the little chips can do practically anything, no matter how complex. However, beneath this complexity is a surprise. The microcontroller operates very simply. This simplicity means that you - the programmer - have to take great pains to create these complex behaviors. Programming requires logical thought and attention to detail.

All programs operate by using a program, or a stored sequence of instructions. These instructions tell the computer what to do. When the computer first starts, it looks at these instructions in sequence. Some instructions read inputs. Others control outputs. Still other instructions do some sort of processing.

The Scenix SX uses a Harvard-style architecture. This means that it has one area where it remembers instructions and another area where it remembers data (including inputs and outputs). This is a common architecture for microcontrollers (although some computers utilize a Von Neumann architecture where data and instructions mix together).

Suppose you started a new job at a factory that makes radios. The plant manager gave you the following instructions:

1. Put an empty crate at the base of the conveyor belt
2. Flip the big red switch on to start the conveyor belt
3. Watch for completed radios to come off the conveyor belt and into the crate. For each radio, click your handheld counter.
4. When the counter reaches 10, flip the switch again to stop the conveyor belt
5. Move the crate and replace it with a new empty crate
6. Reset the counter in your hand
7. Go back to step 2

This is exactly like a computer program. It is a sequence of steps. It has inputs (you deciding that a radio came off the conveyor belt). It has outputs (you flip the big red switch, for example). It also has processing in the form of counting and making decisions. In fact, this is just the kind of job a computer excels at.

Problem #1

There is a slight problem. Outside of Star Trek computers don't understand ordinary instructions like this. How do you instruct the computer to perform these steps? Every computer, from the smallest microcontroller to the largest supercomputer, stores its instructions in the form of numbers. Even worse, computers store these numbers using base 2 arithmetic (binary, a subject covered later in this unit). That means that a computer program looks like a series of 1's and 0's. This is called machine language, and is the basis of every computer program.

Of course, base 2 numbers are not easy for humans to understand, so people usually write the numbers in a more manageable system. However, even then it is hard to comprehend a program written only in numbers. For this reason, engineers typically use some more convenient method of expressing programs.

The most common way to program microcontrollers is using assembly language. This is a short hand method that allows abbreviations to stand in for the 1's and the 0's. You might use instructions like **ADD** or **JMP** (jump). In the old days you'd manually convert this shorthand into 1's and 0's, but today a special program known as an assembler does it for you. Of course, the microcontroller can't run this program, but your PC can. This is often called cross assembling - using one computer to assemble (convert from shorthand instructions to 1's and 0's) code for another computer. The short hand abbreviations, by the way, are known as mnemonics.

Many people find it daunting to program using these low-level instructions. Even though mnemonics are easy to read, they still represent the machine language, which is very simple. For example, the typical microcontroller can't directly multiply and divide numbers. Instead they calculate these operations using addition and subtraction. For this reason, some programmers turn to high level languages like Basic or C - languages you might be familiar with from other computer systems.

If Basic and C are available for microcontrollers, why use assembly language or machine code? The answer is efficiency. Microcontrollers generally have limited amounts of memory. Also, you often need them to perform as fast as possible. A program that uses a high level language will often consume more memory than a well-written assembly program. It may also run more slowly.

If you do use Basic or C, you can count on the major portion of the language to run on your PC. This is similar to cross compiling. You write you C program on the PC and the PC converts your program into machine language. Parallax makes a successful product known as the Basic Stamp that uses the PC to convert Basic code into a quasi-machine language. The Basic Stamp then executes a program that interprets this quasi-machine language to perform the programming steps.

Tip: Different types of microcontrollers have different machine languages. However, most people find that if they learn one microcontroller’s language, others are relatively easy to learn.

Problem #2

The next problem is what to do with the 1's and 0's once you have them. Somehow, you have to move these 1's and 0's into the computer. Older microprocessors used an external memory chip but modern processors have memory on board that you program with a special device known as a programmer. Some microcontrollers require ultraviolet light to erase the memory but the SX is instantly reprogrammable so you don't need to wait for a special light to erase the part.

In a Harvard architecture microcontroller, you can't change the program code while the microcontroller is running. Many microcontrollers can't even read data from their program storage while executing a program. However, the SX has a special feature that allows you to read data from the program's memory while running. This can be useful for storing constants, for example.

Watch Your Language

In this course, you'll use assembly language to program the SX. However, if you are familiar with Basic or C you'll find parallel code examples to help you visualize the assembly code.

The Parallax Basic Stamp uses a particular variant of Basic known as PBasic. The Basic code will mimic the Stamp's language so you can apply the same concepts with the Stamp. There are several variants of the Basic Stamp and one of them has a SX microcontroller in it. However, you must program the Stamp using PBasic -- you can't use machine language. On the other hand, you might wonder why you'd want to use machine language if you could use Basic. The truth is, Basic is great, but some jobs require the speed and capabilities you can only get with machine code.

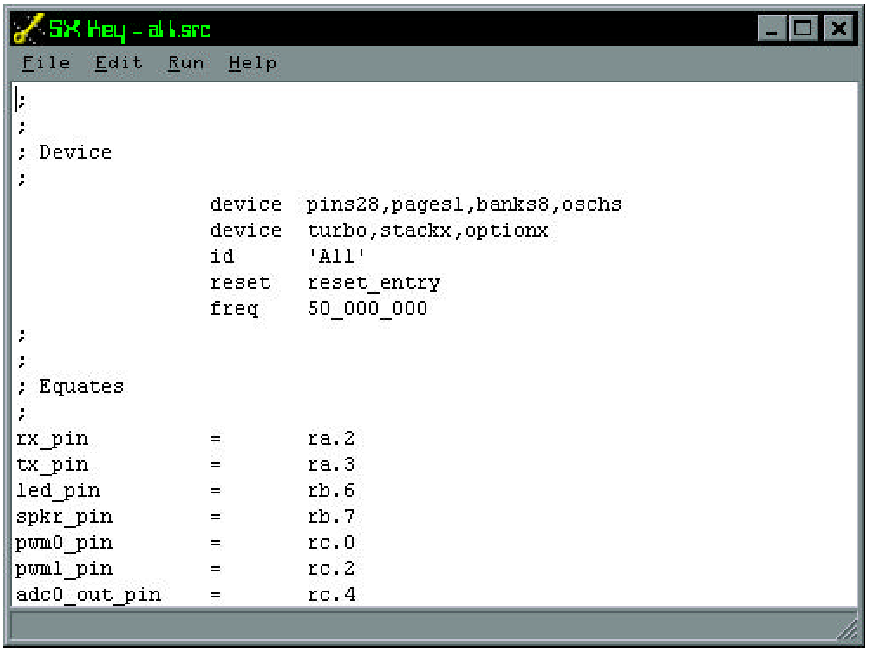

Figure I.1 – The SX-Key Editor

The Working Environment

Figure I.1 shows the main screen of the SX-Key. Looks like a common text editor, and at this point it is. You can enter assembly language code in the window. When you want to test or run your program you can use the Run menu to check your syntax or program the SX chip. To just check your code for simple errors, use the Run|Assemble menu. You can also use Run|Run to execute the code (assuming you have the chip connected to the SX-Key hardware).

Tip: The assemble command only checks for simple syntax errors. Logic errors are up to you to find (with help from the debugger).

Is That It?

The real power of the SX-Key is not entering code. The impressive part is when your code doesn't work. Then you can use the Run|Debug command.

The debugger (see Figure I.2) allows you to watch the SX execute your program one step at a time and examine its internal workings. If you are using the SX-Blitz, you can only program the SX – the Blitz does not support debugging.

The Development Cycle

As you might imagine, such powerful tools greatly simplify programming. However you still need a plan. There is an old saying: "People don't plan to fail, they fail to plan." This is especially true when programming.

Earlier you read that programs read input, process it, and produce output. This is not a bad place to start when designing your software. Complex projects may require more rigorous design techniques, but many times this simple approach is enough. However, nearly every program (especially those for microcontrollers) will follow this model. Identifying your inputs, outputs, and processing is a solid first step towards realizing your design.

The next step depends on your background, experience, and personal preferences. You might start by making a list of instructions similar to the assembly line steps mentioned earlier. Some people prefer to draw the steps of their programs using boxes like a flowchart.

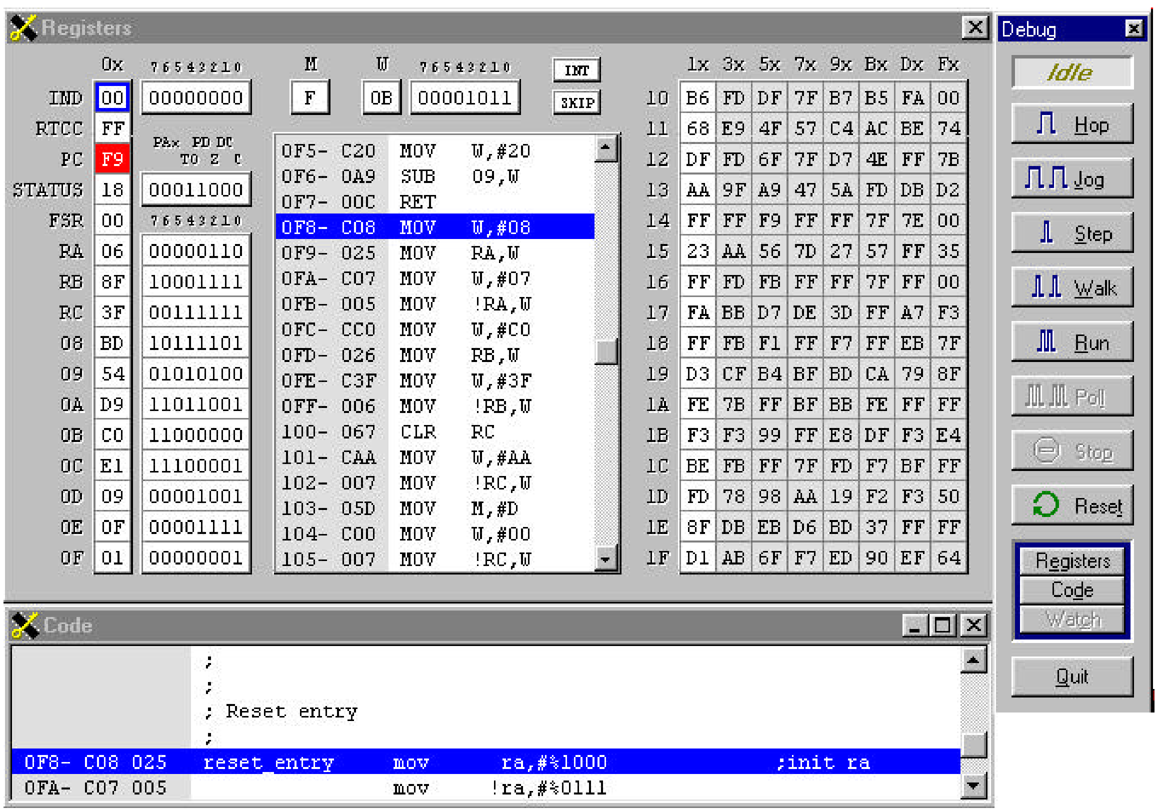

Figure I.2 – The SX-Key Debugger in Action

Once you have an idea of what your program will look like you can make your first pass at entering the program into the SX-Key editor using the assembly language instructions you'll learn in the following units. Your first attempt at running the program might work, but it isn't very likely. When things don't go as planned you'll turn to the debugger for a better understanding of your program's operating.

Even if your program works you may still want to use the debugger to study its operation. Sometimes you will see improvements you missed when thinking about the program in the abstract.

Number Systems

When normal people count they use base 10 or decimal. However, computers like to use binary or base 2. Programmers have to switch between the two and often use other systems as well.

When you say 138 (in decimal) you really mean:

$$1\times100+3\times10+8\times1$$

Decimal digits range from 0 to 9.

Binary numbers are similar, but they use only two digits: 0 and 1. The binary number 1001 is really:

$$1\times8+0\times4+0\times2 +1\times1 = 9.$$

You can see how easy it is easy to convert from binary to decimal. Just remember that each digit is worth double what the digit to the right of it is worth.

Example:

$$10011110 = 2+4+8+16+128 = 158$$

Going the other way is a little more difficult. The trick is to determine which binary digit (known as a bit) is the largest necessary to represent the number. Consider the decimal number 122. The right-most bit in any binary number is always worth 1. The next bit is worth 2 then 4, 8, 16, 32, 64, 128, and so on.

Since 128 is bigger than 122, that bit can't be in the equivalent binary number. By convention, the right-most bit is considered bit 0 and the other bits are numbered sequentially from right to left. So the bit with the value of 128 is bit 7.

However, bit 6, with a value of 64, will have a 1 in the answer since 64 is less than 122. Since 122-64=58 you'll still have to account for this amount. The next bit's value is 32 and 32 is less than 58, bit 5 will also have a 1. The remainder is 58-32=26.

Bit 4 is worth 16 and so it will also be a 1 leaving 10. Bit 3 (8) will also contain a 1 leaving 2. Now consider bit 2. It has a value of 4 but this is greater than the remaining value and so it will contain a 0. The next bit is worth 2 so it will be a 1 and it leaves a remainder of 0. Therefore, all the bits to the left (in this case, only bit 0) will have a zero value.

So the answer is that 122 = 1111010. You can check your work by reversing the conversion. In other words:

$1\times64+1\times32+1\times16+1\times8+0\times4+1\times2+0\times1 = 122.$

It should be obvious, but you can add as many zeros as you like to the left of a binary number (or any number for that matter). So 1111010 and 01111010 and 0000000001111010 are all the same number.

Other Places, Other Bases

Since most people use decimal you have to use it sometimes. But many times it is easier to use other notations that are easier to convert to binary. The most common alternate base is hexadecimal or base 16.

Hexadecimal (commonly known as hex) uses 16 digits -- 0 to 9 and A-F. You can find the values in table I.1. Notice that to convert between binary and hex you can simply use the table. So F3 hex is 11110011 binary.

In hexadecimal each digit is worth 16 times more than the one before. So F3 hex is:

$15\times16+3\times1=243$

And 64 hex is:

$6\times16+4\times1=100.$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Table I.1 – Hexadecimal Digits

Tip: Many calculators, including the CALC program in Windows, can convert between bases automatically.

Say What You Mean

With these different ways of writing numbers, it is easy to get confused. Even the SX-Key assembler can't magically guess which number system you are using. That's why it is important to specify exactly what kind of number you are writing.

To specify the number system in use, you write numbers with special prefixes. A number that begins with a $, for instance, is a hex number. Binary numbers begin with a % character. Since decimal numbers are the most common numbers, they don't have a prefix.

Tip: Not all assemblers use this naming convention. For example, some assemblers use suffixes to indicate the number type. Others use different prefixes. However, the SX-Key assembler you will use in this course will use the prefixes as indicated.

Size Matters

Another concern with numbers is how many bits they occupy. The SX uses an 8-bit word size for data. This is often called a byte. The problem with bytes is that they can only hold numbers from 0-255. What if you need bigger numbers? Or negative numbers? Then you'll need to resort to special techniques found in unit VI.

Remember, by convention, you number bits starting at the right-most bit. So the right-most bit is always bit 0. The left-most bit in a byte is bit 7. This is somewhat confusing because bit 7 is actually the eighth bit (because you started counting at 0 instead of 1).

Incidentally, although the SX uses an 8-bit word for data, its instructions are 12 bits wide. Since the Harvard architecture separates code and data, this isn't a problem, as it would appear to be.

The Hardware Connection

Of course, as nice as the SX-Key is, it is only a means to an end -- programming the actual SX chip!

The SX is an especially speedy processor. It can run at speeds up to 100MHz and can execute most instructions in a single cycle (10nS per instruction). In a real project, you must supply a crystal or a ceramic resonator for speeds greater than 4MHz. However, when working with the SX-Key it provides the clock (you can change the clock speed using the Run|Clock menu).

The SX comes in an 18-pin package and a 28-pin variant. The 18-pin device has 12 I/O pins and the 28 pin device sports 20 I/O pins. Both devices have 2K of program storage and about 136 bytes of data storage (although future devices may have different amounts of memory). There is also a surface mount-only, 20-pin device that is about the same as the 18-pin SX. When you write a 1 to an output pin, it generates (roughly) 5V. If you write a 0 to the pin, it outputs 0V. On input, the pins recognize voltages above a threshold (typically 1.4V) as a 1 and below the threshold as a 0. You can make any pin an input or an output and you can even switch them during program execution.

Obviously, your choice of parts will often hinge on how many I/O pins you need. If you want to use, for example, 4 pins to drive an LCD display, and 8 pins to connect to a keypad, you won't have anything left over for other work if you use the 18-pin SX. However, for this course you may also be constrained by the experiment board you are using since it may only have a socket for one device or another.

You can find the hardware details of the SX in the official data sheet. However, you'll also read more about the SX hardware in the remaining units of this course.

Summary

The old saying goes: "The mightiest oak begins as a tiny acorn." In a similar vein, the simple functions of a microcontroller can build complex systems if you know how to use them.

To understand low-level computers like microcontrollers you have to speak their language -- or at least the shorthand assembly language and hex codes that most people use to represent the arcane machine language.

This unit -- by necessity -- covers these fundamentals. By now you should be itching to really use some hardware. You'll get your chance in the next unit.

Exercises

1. Convert the following numbers to decimal:
   - (a) $27
   - (b) %101110
   - (c) $F1
   - (d) $AA

2. Convert the following numbers to hexadecimal:
   - (a) 100
   - (b) 200
   - (c) 17
   - (d) %10110110
   - (e) %1000001

3. Answer True or False to the following statements:
   - (a) Programs consist of a series of steps.
   - (b) All computers us a Harvard architecture.
   - (c) A Harvard architecture computer uses separate memory for programs and data.

Answers

1. (a) 39 (b) 46; (c) 241; (d) 170
2. (a) $64; (b) $C8; (c) $11; (d) $B6; (e) $41
3. (a) True; (b) False; (c) True

The programs and information in this tutorial are presented for instructional value. The programs and information have been carefully tested, but are not guaranteed for any particular purpose. The publisher and the author do not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors or omissions. The publisher and author assume no liability for damages resulting from the use of the information in this tutorial or for any infringement of the intellectual property rights of third parties that would result from the use of this information.

Rev1.2

Unit II. Your First Program

Unit II from Introduction to Assembly Language with the Scenix SX Microcontroller

© 1999 by Parallax, Inc. All Rights Reserved. By Al Williams, AWC

By now, you are probably ready to jump in and start a project. Good, because that's exactly what you will do in this unit. You should have a PC running the SX-Key software connected to an SX-Tech board. If you don't have an SX-Tech board you can use any other similar development board with some LEDs connected to port B so that they turn on when you output a 0 from the SX (see Figure II.1). Connect an LED at least to two adjacent pins on the SX's B port. If you are industrious, wire 8 LEDs, one to each pin on the B port. You can also save some time if you use LEDs that have internal dropping resistors. These are available from a variety of sources and then you don't have to add the resistor to the circuit. Just wire the LED to 5V and the SX pin.

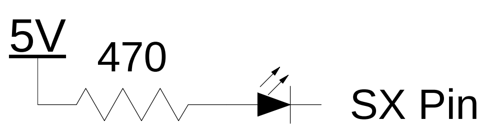

Figure II.3 – LED Circuit

To start with, you'll enter a program into the SX-Key editor, download it to an SX processor, and execute it. You'll see exactly what each part of the program means later in this Unit. For now, just concentrate on getting familiar with the steps involved and your hardware setup.

First Step

If you haven't already, install the SX-Key software as instructed in the manual. The manual will also tell you how to start the program, and you should do so now. The initial screen is blank and you can enter your program here (you can also, of course, load an existing program from disk).

What to enter? That's the problem! For now, enter the following simple program exactly as shown. Note that each line except the one containing **start_point** is indented with a tab. This is a common practice in assembly language – placing labels (like **start_point**) in the first column, and placing commands to the right at least one tab.

```
            device pic16c55,oscxt5
            device turbo,stackx_optionx
            reset       start_point
            freq        50000000   ; 50 Mhz


            org   0

start_point mov   !rb,#0    ; make all of port b outputs
            mov   rb,#0     ; make all port b outputs = 0
            sleep           ; go to sleep
```

By default the SX-Key software is not case sensitive, but you can set it to be if you like (this makes things harder though, so you probably should not do so).

It is a good idea to save your program from time to time. If Windows freezes or crashes for some reason, you'll be glad you saved. Certainly you should save your work before you try to run the code on the SX.

When you finish entering the program, select Run|Assemble from the SX-Key menu. If you entered everything with out any mistakes, you'll see "Assembly Successful" In the status bar. Otherwise, you'll see an error message and the cursor will jump to the line containing the error. Fix the error and try again.

At this point, the only thing the SX-Key software is doing is checking your program for syntax errors. It is still possible (and even likely) to make logical errors that the assembler can't catch. Think about a word processing program's spell checker. It can tell you if you spell 2 as "tew", but it can't warn you if you spelled it as "too" or "to". The assembler has the same problem. It can tell if you've made an obvious mistake, but it can't decide if you're program works as you expect it to operate.

Lock and Load

Once your program assembles correctly, you can download it to the SX chip. The most obvious way to do this is to use Run|Program. This assembles the program again and, if the assembly has no errors, loads the machine code to the SX chip. You can find more about the hardware setup in Appendix B. Of course, if you are using the SX-Tech board, you can also refer to its instructions for the hardware and software setup instructions.

However, you may find it better to use the Run|Run menu item. This works just like the Program command, but it also starts the program running. If you've already used the Program command, you can just use the Run command again, or select Run|Clock to start running.

Either way you start running you should see the LEDs connected to port B light up. Not very exciting, but it is a start. At this point you know your hardware is working and your software is configured correctly.

Tip: Once you program the SX the chip retains the program until you reprogram it.

If you are guessing what the program is doing, you might wonder why the LEDs light up when the pins outputs a zero. This may seem counterintuitive, but it is a common practice. Although the SX can sink and source a considerable amount of current, many chips can sink more than they can source. Because of this, designers often wire LEDs and other loads so that they turn on with a 0 logic level.

So What?

On the face of it, this seems unimpressive. You can make LEDs light up with no external circuitry at all, right? So add the following line of code right before the line that has **sleep** in it:

```
mov   rb,#$AA   ; make every other port b output = 1
```

Now when you run the program, you'll see some lights on and some lights off. Is that correct behavior? After all, the program first turned all the lights on. Then it turned some of them off. Why can't you see all the lights turn on before some of them turn back off? The answer is that the SX chip is running each instruction in 20nS! You'd have to have some pretty good eyes to see those LEDs light up for 20nS.

However, if you could make the SX run instructions one at a time, you could see it. In fact, that is something the debugger can do. Before you dive into the debugger, however, let's take a look at what is happening inside this simple program.

Inside the Program

The easiest way to figure out what this simple program is doing is to examine it line by line. Along the way, you'll see some key concepts that you'll deal with in every program you write. The first two lines begin with the **device** keyword. This is not really an SX command. Instead, it is a directive to the assembler. Most keywords have some equivalent machine language value. However, directives don't generate machine code, they simply give the assembler instructions. In this case we want the assembler to know that we are writing a program where the SX should configure itself to look like a PIC16C55 (an older processor) and a high-speed oscillator (**oscxt5**). The second line informs the assembler that we want to use several special modes that the SX supports. The assembler will use this information to burn the SX configuration fuses. These fuses control the chip's hardware settings and are not part of the actual program. Normally, you will want to replace the **pic16c55** directive with either **sx28l** (for 28-pin devices) or **sx18l** (for 18-pin chips). However, until you learn about banking in unit 6, it is better to stick with the 16c55 emulation.

The next line contains a **reset** directive. This informs the assembler where the program is to start executing. You might think that it would be logical for the program to start at the beginning, but you'll see later that this is not always the case. The name after the directive, **start_point**, is a user-defined label. This label can be any identifier you want and locates a spot in the program.

Tip: Labels and other identifiers can contain up to 32 characters. The first character must be a letter or an underscore. The other characters can be letters, underscores, or digits. You can't use reserved words (like **sleep** and **reset**) as an identifier.

The next line specifies the clock frequency in Hertz. This doesn't really do anything for the SX chip, but it helps the debugger determine what clock frequency you want to use. If you don't specify a **freq** directive, the default is 50MHz. You can also change the clock frequency for running programs using the Run|Clock menu. The assembler allows you to add underscores in any number like this to make it more readable. So you may see a similar line written like this:

```
            freq        50_000_000
```

The next line contains the final directive, **org** (which stands for origin). This directive instructs the assembler to begin generating code at a particular address. In this case, you want to start at the beginning so the **org** is 0.

The next 3 lines (or 4 if you've added the line of code that turns off some LEDs) are the actual program. The things up to this point were simply directives to the assembler. The first program line starts with the **start_point** label. This is so the **reset** directive can refer to it. Notice that the label appears first on the line. The remainder of the line is the actual instructions for the microcontroller.

Registers

The data memory of the SX consists of a small number of byte-sized registers. Although there are well over 100 registers in the SX, your program can only work with 32 of them at a time. In a later unit, you'll learn about banking which allows you to get to all the registers, but for now, suffice it to say that there are 32 registers. Register $08 to $1F are available for you to store data. However, registers $00 to $07 are special because they control the SX chip as your program executes.

For example, register $05 corresponds to the SX's port A. When you read a value from register $05 (known to the assembler as the **ra** register), you are actually reading the digital signals present on port A's input pins. If you write to the **ra** register, you will alter the digital signals that appear on port A's output pins. You can also use $06 (**rb**) or $07 (**rc**). For the 18-pin device (which has no port C) you can use register $07 for data storage if you wish.

This leads to another problem: How do you know which pins are inputs and which are outputs. Initially, all pins are set as inputs. However, your program can change this at any time by storing a special value into the port's direction register. To access the port's direction register you put an exclamation point in front of the register name. Writing a 0 to the direction register makes the corresponding bit an output. A 1 makes it an input.

Now the three lines of the program make more sense. The first line uses the **mov** (move) instruction. This instruction moves a zero into the port B direction register (**!rb**). Notice that the 0 has a # character in front of it. This marks it as a constant. Without this #, the instruction would move the contents of register 0 into **!rb**. You can add a base (or radix) specifier after the #, so #$FF is a hex constant and #%1011 is a binary constant.

The second line uses the same instruction, but now the destination register is **rb** instead of **!rb**. This writes the data out to the port. Since all the pins are outputs, each pin will now have a 0V level. This causes the LED to light.

If you added the extra line of code, it writes $AA to the ports. This is the same as %10101010 so it alternates the LEDs. The final line, **sleep**, shuts the processor down in low power mode. You will rarely use this in a real program – at least, not in this way – most microcontrollers never just stop. Later, you'll see that you might want to sleep until some external event or time period wakes you up, but in this case the processor just sleeps forever – something almost unheard of for a microcontroller.

One other item you might notice in the program is the comments. These start with a semicolon and continue to the end of the line. You can use comments anywhere you want to make notes about the program's operation. This is a good idea in case anyone else has to read your work. It might even help you when you need to review your code 6 months down the road and you can't remember how things worked.

Tip: Another use for comments is to temporarily remove a line from your program. Just put a semicolon in front of the line you want to "delete" and then you can restore it by simply removing the semicolon.

If you were a PBasic programmer, you might like to think of this program as similar to this:

```
DIRL = $FF
OUTL = $00
END
```

Notice that PBasic uses a direction register just like the SX. However, the bit meaning is the opposite. In a Basic Stamp program, direction bits of 0 set input pins, and a 1 sets the output pins.

Taken one piece at a time, this program isn't very complicated at all. However, there is an even better way to understand what it is doing: use the debugger.

Elementary Debugging

Once your program is running, you might like to try executing it with the debugger to see how it works (assuming you are not using the SX-Blitz which does not support debugging). This will also give you practice using the debugger, something you are sure to need before long. To start, use the Run|Debug command. This is similar to the Run|Run command but it also loads a special debugging program into the SX chip. Normally, you don't know this program is present. However, you do have to have some free memory for the debugger or it won't work. In fact, the following requirements are necessary for debugging to work:

- No external clock (the SX-Key supplies the clock)
- Use the **RESET** directive
- No watchdog timer (covered later)
- 2 free instructions in the first bank of program memory
- 136 free instructions in the last bank of memory
- A **FREQ** directive, unless you want to run at 50MHz, in which case **FREQ** is optional

After you press Run|Debug you'll see the usual programming windows. Then you'll see three windows open up. The Registers window contains the contents of the SX registers and a dump of the machine code you are executing. The Code window shows your source code (and the machine code to the left of that). Finally, the Debug window gives you a remote control to start and stop your program in a variety of ways.

The first thing you will notice is that your program starts at the top of memory, not at your starting point. That is because the SX always starts at its topmost address. Since you told the SX it should pretend to be a 16c55, this program begins at address $1FF (this will vary in other modes, but will always be the highest possible address). At this address is a **JMP** instruction that the assembler wrote for you – based on the information in the **RESET** directive. This instruction jumps to a different address and determines the starting address of your program.

In Figure II.2 you'll notice that the Register window has the first 16 SX registers on the left-hand side of the screen. You'll notice the **RA**, **RB**, and **RC** registers, as well as the user registers $08 to $0F. The display is in hex, but directly to the right of each value is the same value in binary. The other registers (in hex only) are on the right-hand side of the Register window.

The center of the screen shows the machine language dump of your program. Notice that some instructions you write in your program actually generate more than one machine language instruction. For example, the line that reads:

```
mov   !rb,#0
```

Really generates:

```
mov   w,#0
mov   !rb,w
```

The **W** register (which appears near the top of the register window) is a special register often known as the accumulator. Practically all math operations occur in the **W** register.

There is no instruction to move a constant into the **!rb** register, so the assembler automatically used the **W** register. This can lead to program bugs if you don't keep it in mind. For example, consider this:

```
mov w,#$AA
mov !rb,#0
; Now w has 0 in it even though you think it has $AA in it!
```

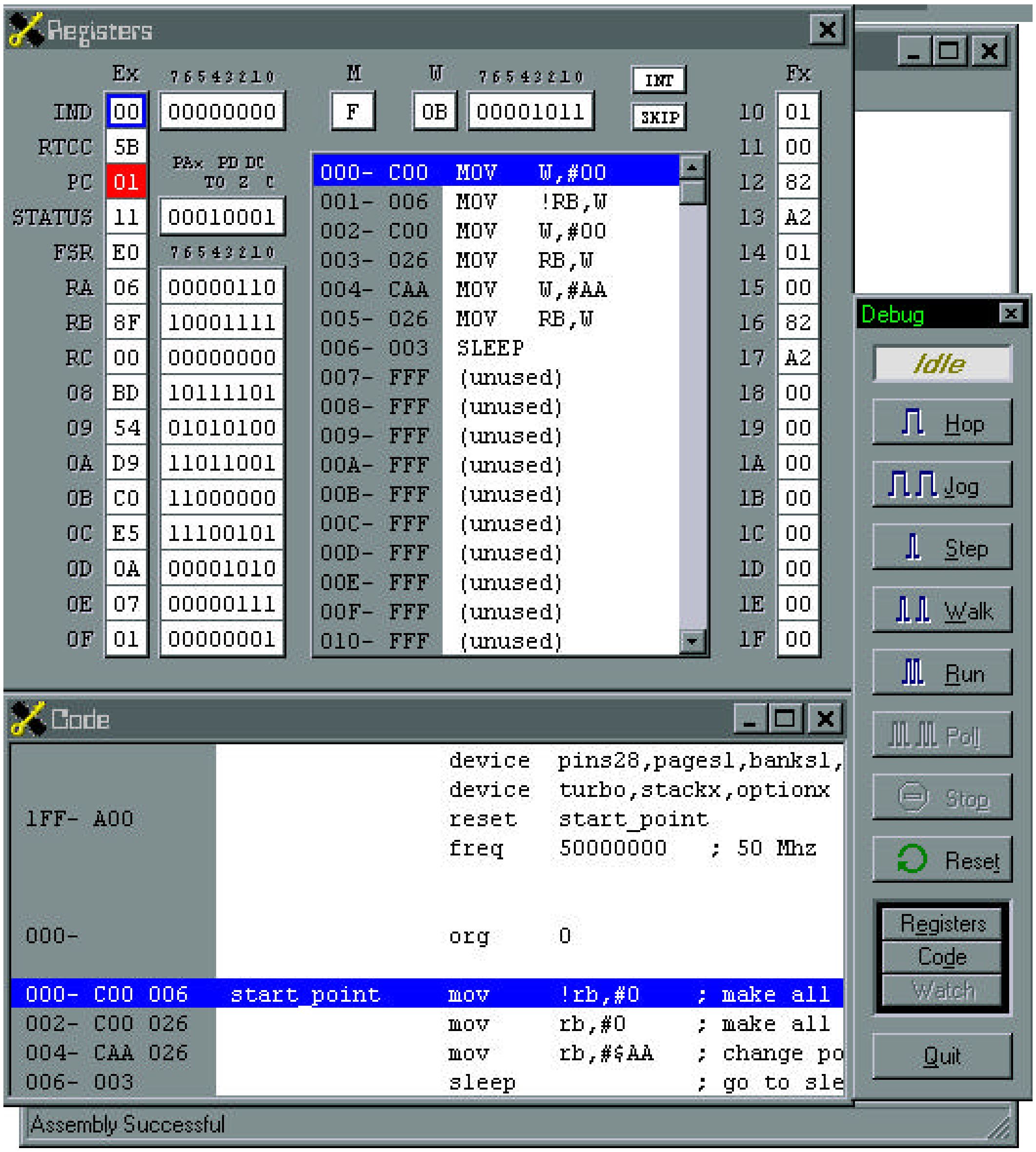

Figure II.4 - The Debugger

The remote control has buttons that you can use to study your program:

- Hop – Executes one assembly language instruction (remember, this might be more than one machine instruction)
- Jog – Executes assembly instructions in slow motion letting you see the results as your program run slowly – press Stop to end Jog mode
- Step – Executes one machine language instruction
- Walk – Similar to Jog mode, but steps machine language instructions instead of assembly language instructions
- Run – Runs your program at full speed. The debugger can't examine registers until you press Poll or Stop
- Poll – This button only becomes active while running. It causes the debugger to freeze the processor momentarily, read the registers so you can view them, and resume program execution
- Stop – End a Jog, Walk, or Run command (only active when these commands are running)
- Reset – Starts the program over

As you step through your program, you'll see a highlight to indicate what instruction your program is executing. Also, registers that change value will appear in red.

Stopping the Debugger

This is a short program, so it is easy to step through it. However, this is not always the case. Many times, the area of your program you want to examine will be buried in the middle of a long program. Perhaps that piece of code only runs when an external event triggers it, or after a time delay. In this case, you'll want to set a breakpoint.

Simply put, a breakpoint is a stop sign in your program. When the SX tries to execute the line of code the breakpoint is on, the debugger takes control and the programs pauses execution. You can resume execution using the Debug remote control, either running the program or stepping through it.

The debugger supports one breakpoint at a time. To set a breakpoint, just click on the line you want to stop at (either in the Register or Code windows). The line will turn red. Now if you press Run (be sure to press Reset first if you've already run the program) the program execution will halt at the breakpoint. Setting a new break point, clears any existing ones. If you want to clear all breakpoints, just click on the red line that already has a breakpoint.

You can also add a breakpoint in your assembly language program so that you'll always have a breakpoint set when you start debugging. You do this by adding a **BREAK** directive in your program like this:

```
mov         !rb,#0
break
mov         rb,#$FF
```

By the way, if you try to set a **BREAK** before a **sleep** instruction it won't work. If you have to do this, just use a **NOP** instruction after the break. **NOP** stands for no operation and the instruction does absolutely nothing but waste time. You may have to use this same trick when debugging code that loops to the same address using **jmp**.

Summary

So far you've read about four instructions, **mov**, **sleep**, **nop**, and **jmp**. There is more to learn about the **mov** instruction, but even then it is obvious you need more instructions to write any sort of useful programs. Still, even this small set of instructions allows us to control the output bits of the SX. In the next unit, you'll learn more about jumps and labels and build more functions into this simple program.

Exercises

1. Since each bit in the direction register stands for a different pin, it makes sense to specify the value for the direction register (and often for the port register itself) in binary. Rewrite the first example program in this unit to use binary numbers instead of hexadecimal numbers.
2. The **JMP** instruction transfers control to a different address. Can you replace the **SLEEP** instruction with a **JMP** back to the top of the program? Predict how this will affect the LEDs.
3. The problem with the program in this unit is that the LEDs change so fast, you can't see them without the debugger. Can you reduce the speed of the SX so you can visualize the LEDs when running without the debugger?

Answers

1. Change #0 to #%00000000 and #$AA to #%10101010
2. Change the **sleep** command to this:
   ```
   jmp start_point
   ```
   The LEDs now change rapidly over and over. You can't really see the lights change, but you'll notice that the lights that turn off appear somewhat dimmer than the ones that are on at all times.
3. Using Run|Clock, you can reduce the clock speed to 400kHz. However, this is still not slow enough to see the LEDs change. Probably the best way to see what the program is doing is to use the Jog or Walk commands in the debugger.

The programs and information in this tutorial are presented for instructional value. The programs and information have been carefully tested, but are not guaranteed for any particular purpose. The publisher and the author do not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors or omissions. The publisher and author assume no liability for damages resulting from the use of the information in this tutorial or for any infringement of the intellectual property rights of third parties that would result from the use of this information.

Rev1.2

Unit III. Simple Flow Control

Unit III from Introduction to Assembly Language with the Scenix SX Microcontroller

© 1999 by Parallax, Inc. All Rights Reserved. By Al Williams, AWC

3

In the previous unit, you wrote and debugged a simple program. This program started at address 0, executed a simple set of instructions, and then went to sleep. While this was good to start with, it is clear that most microcontrollers don't execute a few commands and then stop – they run all the time, monitoring inputs and manipulating outputs.

In this unit, you'll extend the simple program from last time so that it does more interesting things. Along the way, you'll read about a few more simple SX instructions.

Running?

As you ran the last unit's program in the debugger, you may have notice that the **PC** register changed every time you executed a step. If you noticed a little more, you might have realized that the number in **PC** matched the address of the current machine language instruction. That's because **PC** is the program counter. This is a special register that tells the SX which instruction it will execute next.

Do you remember the first instruction you saw in the debugger? It was a **JMP** that the assembler automatically put in at the default reset address so that our program could start where we wanted it to start. Of course, you can also write your own **JMP** instructions to control the flow of execution in your own program. This is similar to using a **goto** statement in Basic or C.

In the last unit's exercises, you changed the **sleep** instruction to a **JMP** to cause the program to restart at the beginning instead of stopping. However, the solution presented isn't as efficient as it could be. Here's the entire solution:

```
                device  pic16c55,oscxt5
                device  turbo,stackx_optionx
                reset   start_point
                freq    50000000   ; 50 Mhz


                org     0

start_point     mov     !rb,#0    ; make all of port b outputs
                mov     rb,#0     ; make all port b outputs = 0
                mov     rb,#$AA   ; change port b outputs
                jmp     start_point
```

What's wrong here? Nothing is actually wrong. However, the code as written keeps storing 0 in the direction register (**!rb**). There is no reason to do this. Once the direction register is set, there is no reason to keep setting it again. It doesn't hurt anything to reset it, but it wastes time that you could use to do something else.

The solution is simple. Just add another label to the line following **start_point**. Call it **again**. Then you can jump to **again** instead of **start_point**. So:

```
start_point     mov     !rb,#0    ; make all of port b outputs
again           mov     rb,#0     ; make all port b outputs = 0
                mov     rb,#$AA   ; change port b outputs
                jmp     again
```

Another way to make the program a bit more readable is to use the **CLR** instruction. The **CLR** instruction can set any normal register or the **W** register to 0. You can't use it with the **!rb** register though. This is also more efficient since using **MOV** to clear a normal register requires two instructions as opposed to a single instruction for **CLR**. Here is the code:

```
start_point     mov     !rb,#0    ; make all of port b outputs
again           clr     rb        ; make all port b outputs = 0
                mov     rb,#$AA   ; change port b outputs
                jmp     again
```

More Interesting?

To make the program more interesting, you'll need a few more instructions. Consider the **INC** (increment) instruction. The **INC** instruction adds 1 to a register. Since the port B pins look like a register (the **rb** register), you can increment it just like any other register.

Change your code to look like this:

```
start_point     mov     !rb,#0    ; make all of port b outputs
again           clr     rb
                inc     rb        ; change port b outputs
                jmp     again
```

What should this do? You'd like the program to cycle the lights in a binary pattern. So first all lights are on, then the LED on pin 0 turns off. Then it turns back on and the LED on pin 1 turns off. Just like counting in binary where on LEDs represent a 0.

That's what you'd like the code to do, but it won't work. Try it. When you run the code, the LEDs seem to stay on all the time. If you single step through the code, you'll see something a bit different. Use the debugger to determine what's wrong with the program (even if you've already figured it out) and then read the next section.

What's Wrong?

As you probably realized, the problem is that jumping to **again** makes the program reset the **rb** register to 0. To fix this problem, move the **again** label to the next line like this:

```
start_point     mov     !rb,#0    ; make all of port b outputs
                clr     rb
again           inc     rb        ; change port b outputs
                jmp     again
```

Now the program works as you'd expect. If you have an oscilloscope, you might find it interesting to watch the port B pins. Bit 0 of port B will generate pulses of a certain width based on the system clock. Bit 1 will emit pulses twice as long. Bit 2 will create pulses 4 times as long, and so on. Using the timings for each instruction provided in the SX data sheets, you can actually calculate these times. The **inc** instruction requires 1 clock cycle (20nS at 50MHz) and the **jmp** requires 3 cycles (60nS at 50MHz). So the pin will change every 80nS. For practical purposes, you've created a square wave oscillator and a divider – all in software.

It is worth noting that the SX has two ways it can execute instructions: compatibility mode, and turbo mode. In compatibility mode, the SX requires more time to execute each instruction. For example, in compatibility mode, an **inc** requires 4 clock cycles. This make the SX compatible with programs written for PIC microcontrollers (from Microchip) that require the slower execution rate. All the programs in this tutorial use the **turbo** clause in the **device** statement, and therefore require about a quarter of the time to execute as they do in compatibility mode. For new programs you'll always want to enable turbo mode so you can get the best possible performance.

When programming microcontrollers, it is often necessary to compute the number of instructions that will execute so you can precisely set times. Sometimes you want to do this to set a time delay. Other times you'll be setting a frequency, as in this case. When you are fine-tuning your delays, you might find the **nop** instruction – the one that does nothing – useful. You first saw this instruction in the last unit. It simply wastes 1 clock cycle.

In PBasic, by the way, you'd use a program similar to this:

```
      DIRL=%11111111          'all outputs
      OUTL=b1
      Loop:
            b1=b1+1
            OUTL=b1
      GOTO Loop
```

One thing to consider is what happens when some of the pins in port B are inputs (which they are not in this case). That could pose a problem since the increment instruction reads the port, increments the value it finds, and then writes the new value back to the port. When some pins are inputs, the instruction will read the input pins correctly and they will reflect the external stimulus placed on the SX chip. When you increment that, you may or may not get what you expect.

As an example, suppose that bit 7 was an input. When you write 0 to the port, that has no effect on bit 7. If port B's pin 7 has a logic low applied to it, the first **INC** instruction will work as you'd expect. It would read a 0 and write a 1 to the output. However, if the pin were high, the **INC** instruction would read a %10000000 and write %10000001. This probably wouldn't hurt anything, there are cases where this is a problem. Always be wary of using instructions that read, modify, and write on I/O port registers.

Other Forms of JMP

The **jmp** instruction, by the way, has two other forms that you can use. One is that you can use the **W** register (the accumulator) as the destination address. Just write:

```
JMP         W
```

This is useful when you want to use a calculation to determine where to jump. The other form of **JMP** isn't a **JMP** at all. The **ADD** allows you to add the **W** register to the **PC** register. This causes a jump over a certain number of instructions. Of course, the **ADD** instruction really just adds the **W** register to any other register. It just so happens that changing **PC** is effectively a jump. For example, consider this:

```
CLR         8     ; clear register 8
MOV         W,#2
ADD         PC,W
INC         8
INC         8
INC         8
BREAK             ; what is in reg 8 now?
INC         8
```

When the debugger hits the breakpoint, register 8 contains 1 because changing **PC** causes the first two **INC** instructions to not execute. The assembler allows you to write this instruction as **JMP PC+W** to make your program easier to read.

Tip: Since the 2 in this example is a constant, you really could use a regular **JMP** instruction to skip these two instructions. One way, of course, would be to label the target of the **JMP**. However, you can also use the special label **$** which means the current address. So you could write **jmp $+3** instead. Why +3 instead of +2? Since $ refers to the current address, you have to add 1 just to get to the next instruction. Adding 2 would only skip 1 instruction.

The real value to using **ADD** to perform a jump is when you compute the offset at run time. This allows you to create data and jump tables as you'll see later in this tutorial.

In this example, using 8 as a register number is confusing. Remember, it isn't a constant because it didn't start with a #. However, it is much nicer to name your variables in a meaningful way. The assembler provides you a couple of ways to do this that you'll read about in the next unit.

Of course, sometimes you want to jump only if some condition is true of false. For example, you might want to jump only when the user presses a button, or when a sensor reads a certain value. You'll find out how to do that in Unit V.

Local Labels

One challenge when you are programming is coming up with new names for every label. The SX-Key assembler lets you create local labels that begin with a colon. These labels are only valid in between normal (or global) labels. Because the local labels are only valid within global labels, you can define the same label more than once without confusion. Consider this:

```
top         mov         w,#0        ; top is a global label
:loop      .                        ; the first loop
           .
           .
            jmp         :loop       ; goes to first loop
ok          mov         w,#9        ; ok is a global label
:loop      .                        ; the second loop
           .
           .
            jmp         :loop       ; jumps to second loop
```

You never have to use local labels. However, using them can make your life easier and your code more readable. The alternative is to generate unique labels for every address of interest in your program.

Another Way to INC

Sometimes you'd like to increment the value in a register, but you don't want to return the value to that register. In this case you can use a special form of the **mov** instruction:

```
mov w,++8
```

This leaves the result in the **W** register and does not change register 8. This allows you to use the register in other calculations without disturbing it.

In general, math operations always have these two forms. For example, the opposite of incrementing is decrementing (**dec**). This instruction subtracts one from a register. You can write it as:

```
dec 8
```

or:

```
mov w,--8
```

The first form subtracts one from register 8 updating the value. The second form does the subtraction but leaves the result in **W** without changing the original value.

Tip: Basic has no exact analog to **inc** and **dec** (other than **x=x+1** or **x=x-1**). However, in C, you can think of **inc** and **dec** as the ++ and – operators, respectively.

Stopping the Processor

In the early examples, the program used the **sleep** instruction to halt the processor. This might not seem very practical, but there are a few places where it can come in handy. For example, imagine a microcontroller that dials an emergency phone number. The signal to begin could be applying power to the circuit. The program would dial the number and then go to sleep, waiting for another power cycle to run again.

However, the main reason you'll use the **sleep** instruction is to put the processor in low-power mode until some external event occurs or some time period elapses. External events usually take the form of interrupts, a topic that will wait until Unit VII. However, you can wake up at a predetermined time by using the watchdog timer. The main purpose of the watchdog timer is to reset the processor in the case of a malfunction. However, you can also use it as a timer to set a wake up time.

About the Watchdog

To enable the watchdog, add the **watchdog** setting to the **device** statements near the beginning of the program. Notice that turning on the watchdog will prevent the debugger from operating correctly, however. The idea behind the watchdog is that your program should use the **clr** instruction to zero the **!WDT** register periodically. This indicates that the program is working. If you fail to clear this register after a certain period of time, the processor resets.

Tip: The usual purpose of the watchdog timer is to reset the processor in case of a failure. It is usually best to have a single point in your program that clears the watchdog timer (**WDT**). That way the chances of your program crashing and still clearing the timer are remote. If your program stops behaving correctly, the watchdog timer will restart it.

How long is that period? The SX has an internal oscillator for the watchdog that nominally runs at 14kHz and the watchdog times out after 256 counts. So the timeout period is about 18mS. So if you issues a **CLR !WDT** instruction at least once every 18mS, you won't get a watchdog reset.

For timing purposes, this might not be long enough, however. The SX allows you to further scale the watchdog timer by setting bits in the **!OPTION** register. In particular, bit 3 of this register is set to 1 if you want to use the prescaler with the watchdog timer. Bits 2, 1, and 0 set the divide rate (see table III.1). The highest divide rate is 1:128 so the maximum time out is about 2.3 seconds.

Bit 2	Bit 1	Bit 0	Divide Rate
0	0	0	1:1
0	0	1	1:2
0	1	0	1:4
0	1	1	1:8
1	0	0	1:16
1	0	1	1:32
1	1	0	1:64
1	1	1	1:128

Table III.1 – Watchdog Timer Prescale Values

How can you set a single bit in a register? You can use **SETB** to set a bit to 1 and **CLRB** to clear a bit to 0. So to turn on the watchdog prescaler and set the divide rate to 1:32 you could write:

```
setb !option.3
setb !option.2
clrb !option.1
setb !option.0
```

The advantage to doing this is that you don't disturb the rest of the register. However, it is also possible to observe that the defaults for the remaining bits of the **!option** register should be 1's. So if you knew you wanted 1's in the other positions, you could write:

```
mov !option,#$FD
```

The **!option** register defaults to all 1's anyway, so if you want the maximum time out value, you don't need to do anything but enable the watchdog timer. Consider this program:

```
                device  pic16c55,oscxt5
                device  turbo,stackx_optionx, watchdog
                reset   start_point
                freq    50000000   ; 50 Mhz

                org     0

start_point     mov     !rb,#0    ; make all of port b outputs
agn             mov     w,#$FF
                xor     8,w       ; invert bits
                mov     rb,8
                sleep
```

This program will cause the LEDs to flicker so that you can actually see them. The only problem is that you don't know which LEDs will be on and which will be off initially. The program uses the **xor** instruction to exclusive-or the contents of register 8 with the constant $FF. You'll read more about **xor** in the next unit, but for now just realize that these two instructions will flip all the bits in register 8. That is to say that all 0s in register 8 will become 1s and all 1s will become 0s. You can, by the way, replace the **mov** and **xor** instructions with the **not** instruction which also flips the register bits and takes less time to execute. For now, however, leave the code as it is because the next until will use the $FF constant to demonstrate some important ideas.

The last thing the program does is to store register 8's contents into port B. Since the code just flipped all the bits, all the LEDs that were on will turn off and all the ones that were off will turn on. Then the SX goes to sleep. However, since the watchdog is on (notice the **watchdog** clause in the second **device** line) the processor will reset in about 2.3 seconds. This will then flip the bits in register 8 again, reversing the state of the LEDs. Don't forget that you can't debug this program because it uses the watchdog. You'll have to use the Run | Run command to see the program work.

Earlier, you read that programs that use the watchdog must use **clr !wdt** to reset the timer. This program, however, doesn't clear the watchdog. Why? Because this program deliberately wants the watchdog timer to reset – that is how the program delays long enough for the LEDs to blink.

Of course, it would be nice to know that the reset was from the watchdog timer. You can do this by examining the bits in the **status** register. In particular, bit 4 will be 0 if the watchdog triggered a reset. If bit 3 is a 0, then a **sleep** instruction was active at the time. If you knew how to test these bits (a topic coming up shortly) you could initialize register 8 to a known value when a real reset occurred and not initialize it when a watchdog reset occurred.

Using the watchdog for timing is a bit unusual, but perfectly legitimate. In later units you'll find two other ways to make time delays: programmed loops and using the real time clock. These will be easier, because you'll be able to use the debugger when you employ these methods. Another advantage: when the processor resets, there is a brief time that all pins return to the input state until your program sets the direction register. The other methods for generating a time delay allow your program to stay in control of the processor at all times.

Summary

This unit covers a lot of instructions including:

- **jmp** – Jumps to a new program location
- **sleep** – Stops the processor
- **inc** – Adds 1 to a register (also use **mov w, ++r** to put result in **w**)
- **dec** – Subtracts 1 from a register (or use **mov w, --r**)
- **nop** – Does nothing for 1 clock cycle
- **setb** – Sets a bit in a register
- **clrb** – Clears a bit in a register
- **clr** – Sets a register, **w**, or the watchdog timer to zero
- **not** – Inverts bits in a register

· **xor** – Exclusive-ors the bits in a register (more in the next unit)
· **add** – Adds **w** to a register (more in the next unit)

You also read about the **PC** register, and parts of the **!option** and **status** register. In the next unit, you'll find out even more about arithmetic and variables, paving the way for more powerful programs.

Exercises

1. If you have access to an oscilloscope, add some **nop** instructions to the programs that blink the LEDs and examine the results.
2. Modify the watchdog program so that the LEDs blink at one half of the original rate (about 1.15 seconds).
3. What if you wanted to stop the watchdog LED program without using **sleep** and without triggering a watchdog reset? Modify the code so that it halts and does not reset. This will result in a steady pattern of LEDs lighting.

Answers

1. Here is an example of the solution:
```
start_point     mov     !rb,#0    ; make all of port b outputs
                clr     rb
again           inc     rb        ; change port b outputs
                nop               ; add more nops if you want
                jmp     again
```

2. To modify the rate of blinking, you'll change the watchdog timer prescaler value. One way to do this is to place **mov !option, #$FC** near the beginning of the program. You can also use **setb** and **clrb** to set and clear the individual bits in the **!option** register.
3. Replace the **sleep** instruction with:
```
      halting clr !wdt
              jmp halting
```

The programs and information in this tutorial are presented for instructional value. The programs and information have been carefully tested, but are not guaranteed for any particular purpose. The publisher and the author do not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors or omissions. The publisher and author assume no liability for damages resulting from the use of the information in this tutorial or for any infringement of the intellectual property rights of third parties that would result from the use of this information.

Rev1.2

Unit IV. Variables and Math

Unit IV from Introduction to Assembly Language with the Scenix SX Microcontroller

© 1999 by Parallax, Inc. All Rights Reserved. By Al Williams, AWC

The SX uses its registers as data storage. In the examples from previous units, we have simply referred to registers by their numbers. Remember, the first seven or eight registers (depending on the exact processor type) have special names (like **rb**, **status**, or **!option**) and functions.

The special names of these registers help you remember what they do. How can you use meaningful names for registers that you use?

Suppose you want to use register 8 as a variable in your program. There are several ways you can do so. First, you can set up an equate in one of two ways. Near the top of the program you could write:

```
Myvar EQU 8
```

Or:

```
Myvar = 8
```

Now you can replace all the occurrences of 8 with **Myvar**. You can use this method to define any constant even if it is not a register number. The assembler simply replaces every occurrence of **Myvar** with 8.

The other way to define a variable is by reserving space for it using the **DS** directive. The **DS** directive usually has a label in front of it, and has the number of bytes to reserve following it. So to replace the above equates with a **DS** directive you could write:

```
       org 8
Myvar ds 1
```

The confusing part about this is that the **org** directive can refer to the data space or the program space, depending on the context. In this case, the 8 refers to the data memory. Before you start writing program steps, you'll want to write another **org** directive to set the beginning of your program (often location 0).

It is perfectly normal to specify several variables one after another. For example, consider this code that declares a byte variable named **Abyte** and two bytes named **Tbytes**:

```
      org 8
Abyte  ds 1
Tbytes ds 2
```

When you use a variable name in your program, the name of a multi-byte variable refers to the first byte of the variable. So consider this statement:

```
       mov w,Tbytes
```

This loads the first byte of the variable into w. On the other hand, look at this line:

```
       mov w, Tbytes+1
```

This line of code will access the second byte. Is this any different than the following program snippet?

```
      org 8
Abyte  ds 1
Tbytes ds 1
Tbyte1 ds 1
```

No. There is no difference except that using this form, you can use **Tbyte1** instead of **Tbytes+1**. Of course, you can still use **Tbytes+1**; the assembler does not care.

An Example

Remember the blinker programs in the last unit? Here it is again:

```
      device  pic16c55,oscxt5
      device  turbo,stackx_optionx, watchdog
      reset   start_point
      freq    50000000   ; 50 Mhz

                org     0

start_point     mov     !rb,#0    ; make all of port b outputs
agn             mov     w,#$FF
                xor     8,w
                mov     rb,8
                sleep
```

Here is the same program using symbolic variable names:

```
                device  pic16c55,oscxt5
                device  turbo,stackx_optionx, watchdog
                reset   start_point
                freq    50000000   ; 50 Mhz
```

```
                org     8                  ; data start
outval          ds      1

ledport         =       rb
flipmask        equ     $FF

                org     0                  ; code start

start_point     mov     !ledport,#0        ; make all of port b outputs
                                           ; changed to use single
                                           ; instruction to xor with
                                           ; constant
agn             xor     outval,#flipmask
                mov     ledport,outval
                sleep
```

Notice that you can use equates to redefine the standard symbols. This program uses both = and **EQU**. This is often a matter of personal choice. However, once you define a symbol with **EQU** you can't change it later during assembly. Defining a symbol with = allows you to change it later. In this program, like most simple programs, the symbol values don't change at all, so you can use either method.

Tip: When you define a symbol for a constant (like **flipmask**) it still requires the # character to precede it. Without it, the assembler will think you are defining a register number.

Another way to use an equate is to define a name for a particular bit. You can specify bits in SX assembly language using a period after the name of the register and then the bit position. For example, the least-significant bit in register **rb** is **rb.0**. The most significant bit is **rb.7**. Using an equate you can define a meaningful name to bits:

```
LEDpin              equ   rb.0
```

Using names for the registers and constants make the program much more readable. It also allows you to easily change things if you want. For example, it would be simple to change this program to blink LEDs on port A instead of B. It would also be no trouble to change the register from register 8 to another register, if you wanted to do so.

Assignment

In Basic or C, you can assign one variable to another. The SX can do this too using the **mov** instruction. For example:

```
      org   8
byte1 ds    1
byte2 ds    1

      org   0
      mov   byte1,#$AA
      mov   byte2,byte1
```

This piece of code will put $AA in **byte1** and then put the contents of **byte1** into **byte2**.

Tip: The SX machine language does not really have an instruction that moves one register to another. That means the assembler generates a two-part instruction for the second **mov** instruction in this program. The two instructions are actually:

```
      mov   w,byte1
      mov   byte2,w
```

So this one line of code does destroy the **w** register. This can also lead to inefficiencies. For example, consider this:

```
      mov byte2, byte1
      mov byte3, byte1
```

This code unnecessarily loads the **w** register twice. A better way to do this would be:

```
      mov byte2, byte1
      mov byte3, w
```

Or:

```
      mov w, byte1
      mov byte2, w
      mov byte3, w
```

Both of these take 3 instructions (instead of 4) and execute more quickly than the first example.

The only problem is with multi-byte variables. The SX only deals with bytes. That means that if you want to work with larger quantities, you'll have to break up the operations byte by byte. For example, you'd need two **mov** instructions to copy a two-byte variable to another two-byte variable.

For now, stick to bytes. However, bytes can only store numbers ranging from 0 to 255 (or –128 to 127). So if you need numbers larger than this, you'll have no choice but to resort to larger variables.

Performing Math

In the last unit, you saw that the **add** instruction can add the **w** register and another register. You can leave the result in **w** or in any register you like. You can also add a literal to a register, or two registers together. However, these are two instruction sequences that destroy the **w** register in the process. Here are some examples:

```
      org 8
avar  ds 1
bvar  ds 1

      org 0
      .
      .
      .
      add w,avar        ; w=w+avar
      add avar,w        ; avar=w+avar
      add avar,#10      ; avar=avar+10 (w destroyed)
      add avar,bvar     ; avar=avar+bvar (w destroyed)
      add bvar,avar     ; bvar=avar+bvar (w destroyed)
```

The byte-size of these operations can lead to a problem. What happens if the answer is larger than 8 bits? For example, if **w** contains $FF and you add **w** to a register that contains $10, what happens? The answer is that the SX truncates the result. However, to let you know that this has happened, it sets the carry flag (bit 0) in the **status** register. This is true regardless of the destination of the answer. Another bit in the **status** register (bit 2) is set whenever the answer is zero. You can use **status.0** and **status.2** to refer to the carry and zero flags, or you can use the symbolic names, **C** and **Z**.

Later in this unit, you'll learn how to examine these flag bits and use them to perform multi-byte math. You should be aware that not all operations affect these flag bits in the same way. For example, the **inc** and **dec** instructions (covered in the last unit) add or subtract 1 from a register. However, they do not set the carry flag. They do, however, set the zero flag. The SX data sheet tells you which flags each instruction affects.

The opposite of adding, of course, is subtracting. The **sub** instruction can subtract **w** from any register. The result remains in the register. If you want to put the result in **w**, you can use this form of the **mov** instruction (where **R** is the register you want to use):

```
mov w,R-w   ; w=R-w
```

You can also subtract two registers or a literal from a register. However, both of these are really two machine language instructions and destroy **W**. So:

```
sub avar,W
sub avar,#100     ; avar=avar-100 (w destroyed)
sub avar,bvar     ; avar=avar-bvar (w destroyed)
```

The carry flag (bit 0 of **status**) has reversed meaning for **sub**. Suppose you subtract 100 from 30. The carry flag will be clear to indicate that the subtraction underflowed. However, if you subtract 30 from 100, carry will be set indicating that the subtraction yielded the correct result. Subtracting also affects the zero flag.

If you can add and subtract, you might wonder about multiplying and dividing. Simple microcontrollers like the SX can only add and subtract. However, using some techniques you'll see in the next unit, you can decompose multiplication and subtraction into multiple additions and subtractions.

Two's Compliment Numbers

If the carry flag is clear after subtraction, does that mean that the answer is incorrect? Not necessarily. Any microcontroller, including the SX, can handle negative numbers by using two's compliment arithmetic. The idea is simple. Treat the topmost bit (bit 7, in this case) as a sign bit. If the bit is 0, then the number is positive. If the bit is 1, then the number is negative. To represent a negative number, invert it and add 1. Obviously, to find out what number a negative number is, you'd subtract 1, and invert it again.

Consider what happens if you subtract 60 from 40. The correct answer, of course, is negative 20. The SX, however, returns %11101100 ($EC). If you invert this number (%00010011) and add 1 (%00010100) you'll find the result is in fact 20. You can also make up new negative numbers. Suppose you want to add –5 to 10. First, find the binary representation of 5 (%00000101) and invert it (%11111010). Next add 1 to get %11111011 ($FB or 251). If you add 10 to 251, you get 261. But the SX does not get 261! It truncates the result to 5 (the bottom 8 bits of $105). Of course, 10 + -5 is 5, so the answer is correct.

These operations, by the way, are easy to perform on the SX. The **not** instruction will invert bits and **inc** or **dec** will add or subtract 1. So handling these negative numbers is not too difficult even at run time.

The downside to two's compliment math? It limits the numbers you can represent. For a byte, the numbers between 0 and $7F represent 0 to 127 and the numbers from $80 to $FF represent –128 to –1.

More Carry Tricks

Suppose you need larger numbers, say 0 to 999. You'll need to use more than 1 byte. A two-byte number can hold from 0 to 65535, plenty of room for this job. The problem is, how do you do math with these larger numbers.

The **addb** and **subb** instructions will add or subtract a bit – which could be the carry bit – from a register. Consider this simple program:

```
                device  pic16c55,oscxt5
                device  turbo,stackx_optionx
                reset   start_point
                freq    50000000   ; 50 Mhz

                org           8         ; data start
counter         ds            2


                org           0         ; code start

start_point     clr           counter
                clr           counter+1  ; clear both bytes
again
                                          ; do a 16-bit add
                add           counter,#1
                addb          counter+1,status.0
                jmp           again
```

Here, the code is adding 1 to the 16-bit variable **counter**. It also adds the carry bit to the top 8 bits of the counter. Since the carry bit will only be set when the counter overflows, the count will be correct. You can do the same thing with subtraction by using **subb** instead of **addb**.

By using more registers and more **addb** or **subb** instructions, you can manipulate numbers of arbitrary size. A 24-bit number (3 bytes) can hold up to around 16 million. A 32-bit number (the same size the Pentium PC uses; 4 bytes) can hold numbers of around 4 billion in value.

Try It!

Enter the code above and step through it. You'll quickly get tired of watching register 8 cycle endlessly upwards. The Jog command helps, but it still takes a while to get to the interesting part of the code. This is a good time to learn a few extra features of the debugger. First you can click on the box for register 8 and change the value of the register. So if you plug in $FE (or %11111110) in the register 8 box, you'll be much closer to seeing the roll over! This works for all of the registers visible in the debugger.

Another annoyance is that you have to know that the counter variable is actually in registers 8 and 9. An easier way to observe the contents of memory is to use a **watch** directive. This is a statement in your program that tells the debugger to display a piece of memory with a name, and to format it so that it is meaningful. You specify the memory location, the size of the variable, and the format you want. For this program, try adding this line somewhere in your file:

```
watch counter,16,UDEC
```

This will show the 16-bit variable at location **counter** as an unsigned decimal number. You can find a list of all the format codes in Table IV.1.

Format Code	Appearance
UDEC	Unsigned decimal
SDEC	Signed decimal
UHEX	Unsigned hex
SHEX	Signed hex
UBIN	Unsigned binary
SBIN	Signed binary
PSTR	Fixed-length string of ASCII characters
ZSTR	String of ASCII characters terminated with a zero

Table IV.2 - Watch Format Codes

Tip: ASCII (American Standard Code for Information Interchange) is a way to represent text characters as a 7 or 8 bit number. For example, in ASCII, an A is $41, a blank is $20, etc.).

A Few More Functions

You'll often use the carry for a variety of functions. Earlier in this tutorial, you read that you can use **setb** and **clrb** to set and reset a bit. Since the carry bit is just a bit in the **status** register, you can use these instructions to affect the carry.

However, this is a frequently used function, so the assembler provides other instructions to do it so you can type less. In particular, **clc** clears the carry (**clz** clears the zero flag) and **stc** sets the carry (**stz** sets the zero flag).

The real trick is to control your program's flow based on these flags. There are several ways to do this. First of all, the generic **jb** instruction will execute a jump if the specified bit is set. So to jump to **lbl1** if the carry flag is set, you could write:

```
jb status.0,lbl1
```

Of course, using **jb** you can specify any bit. However, the carry and zero bits are very common bits to test, so the assembler also allows you to use the **jc** and **jz** instructions to test for the carry or zero conditions. You can also use **jnb** (or **jnc** or **jnz**) to jump when the bit is clear instead of set.

By performing a subtraction and then testing the carry and zero flags, you can easily write programs that can tell if one number is greater than, less than, or equal to another number. For example, suppose you wanted to know if variable **x** was greater than variable **y**:

```
mov w,x
mov w,y-w
jnc x_greaterthan_y
```

This works because subtracting **x** from **y** will only be negative (that is, cause an underflow) if **x** is greater than **y**. Remember that carry is clear on an underflow when subtracting.

Tip: You might consider computing **x-y** and changing the **jnc** to **jc**. That would also work, but it would jump if **x** were greater than or equal to **y**. To see why, work out the case where **x** is equal to **y**. Of course, you can use **jz** to test for equality and **jnz** to test for inequalities. See Table IV.2 for a summary of possible results when subtracting two numbers.

Carry	Zero	Meaning
0	0	a<b
X (don't care)	1	a=b
1	0	a>=b

Table IV.3 - Results When Computing A-B

Testing for equality with zero is a very common operation, so the assembler lets you write it in a special way. You can use **test**. The **test** instruction sets the zero flag based on any register (including the **w** register).

Another common function relating to zero testing is to increment or decrement a register and jump if the result is zero. You can use **djnz** (to decrement) or **ijnz** (to increment) for this purpose.

Here is another LED flasher that uses **djnz** to blink the LEDs a total of ten times:

```
                device  pic16c55,oscxt5
                device  turbo,stackx_optionx
                reset   start_point
                freq    50000000   ; 50 Mhz

                org     8         ; data start
counter         ds      1
pattern         ds  1
                watch counter,8,udec
```

```
                      watch pattern,8,ubin

                      org     0           ; code start

start_point           mov     !rb,0       ; all outputs
                      clr     rb          ; all low
                      mov     counter,#10 ; 10 times
again
                      mov     rb,pattern
                      not     pattern
                      djnz    counter,again
                      sleep
```

Notice that the blinking code executes 10 times because the **counter** variable starts with 10, and reduces by 1 until it reaches zero. This is a powerful idea and often used in computer programs. Code like this is known as a loop because it executes in a loop as often as you need.

In Basic or C, you'd do something like this with a **for** statement. In Basic, for example, I might write a loop as:

```
FOR counter = 10 to 1 step –1       ' Do the work
NEXT
```

Of course, you'd usually see this reversed, with **counter** ranging from 1 to 10. You could do this too, but it takes a few more assembly language instructions:

```
inc counter    ; assume counter was set to 0 at beginning
mov w,#10
sub w,counter-w
jnz again
```

In the next unit, you'll se a series of compare instructions that can perform this type of logic in one assembly language instruction (but they just write the same sort of code you see above).

Programmed Delays

Another important use of loops is in developing programmed delays. In the previous unit, you saw how to use the watchdog timer as a crude timing device. However, this is not the ideal way to generate a time delay. The watchdog timer makes it hard to debug your program since the SX-Key can't debug your code with the watchdog set. Also, the watchdog can't generate arbitrary delays, and you lose control of the program while waiting for the delay.

However, if you know your clock speed, and the number of cycles each instruction takes, you can compute loops that will cause the appropriate delay. For example, suppose you wanted to generate a 1kHz tone. A 1kHz tone cycles every 1mS (1/1000 = .001) so to make a 1kHz square wave, the SX needs to turn a pin on, wait for 500uS (half of 1mS), turn the pin off, wait another 500uS, and then start over.

Assume you have a piezoelectric speaker connected to pin 7 of port B (a piezo speaker has a high-impedance and you can drive it directly from the SX's output pins). If you could toggle pin 7 at this rate, you'd hear a 1kHz tone coming from the speaker.

The problem is that 500uS is an eternity for the SX. At 50MHz, each instruction cycle (in turbo mode) takes 20nS. So to pause 500uS you'll need 25000 instructions cycles! Consider this simple loop:

```
      clr   delay
wloop djnz  delay,wloop
```

Studying the SX data sheet, you can find that the **djnz** instruction takes 4 cycles every time it has to jump, and 2 cycles if it doesn't have to jump. The **clr** instruction takes 1 cycle. So the total number of cycles in this loop is 256 * 4 + 3 or 1027, a far cry from the 25000 you need. Of course, you could use a 16-bit delay, but this is hard to calculate since the total time through the loop varies depending on the carry flag's status. Instead, it is usually simpler to place this loop inside another loop. Dividing 25000 by 1027 you'll find you need about 24 repeats of this loop to get to 25000. So:

```
      mov   delay1,#24
oloop clr   delay
wloop djnz  delay,wloop
      djnz  delay1,oloop
```

Of course 24 * 1027 = 24648, not exactly the right answer. However, the outer loop adds 95 cycles to the total loop (see if you can calculate that number). That brings the total delay to 24743 (a 1.02% error). For many purposes, this is not a problem. If you needed a more exact figure, you could reduce the number of cycles in the inner loop and increase the count in the outer loop until you get as close as necessary. You can also adjust the timing of the loops by adding **nop** instructions inside the loop to stretch it out.

Logical Functions

Since microcontrollers and other computers work with binary, it isn't surprising that they contain many operations designed to operate on the bits of word. Like other operations, these work on the **w** register and an arbitrary register with the result going to the register of your choice. You can also use a register and a constant, or two registers, but if you do, you will generate more than one machine language instruction and destroy the **w** register in the process. The main logical functions include **and**, **or**, and **xor**.

What do these functions do? They simply examine the two values you supply bit by bit and generate an output bit base on the corresponding input bits. Take **and**, for example. If you use **and** on %10101010 and %11110000, the result is %10100000. Why? Because **and** only outputs a 1 if both input bits are 1. The **or** instruction outputs a 1 if either input bit is 1. The **xor** instruction outputs a 1 if either input is a 1, but not if both inputs are a 1. You can find a summary of these operations in Table IV.3.

Instruction	Truth Table Input	Input	Output	Move to W Form
And	0	0	0	and w,R
	0	1	0	
	1	0	0	
	1	1	1	
Or	0	0	0	or w,R
	0	1	1	
	1	0	1	
	1	1	1	
Xor (exclusive or)	0	0	0	xor w,R
	0	1	1	
	1	0	1	
	1	1	0	
Not	0		1	mov w,/R
	1		0	
RL (rotate left)	n/a			mov w,<<R
RR (rotate right)	n/a			mov w,>>R

Table IV.4 – Logical Instructions

You've already seen that you can use **not** to invert the bits in a register (including the **w** register). You can also rotate or shift bits left or right by using **rl** (left) and **rr** (right). Unlike the other logical instructions, these commands operate on a single register (or the **w** register in the case of **not**). When you shift a register left, each bit is replaced by the bit prior to it. So bit 7 gets the value of bit 6, bit 6 gets the value of bit 5, and so on. Bit 0 gets the value of the carry flag and the carry flag's value gets set to the original value of bit 7. Shifting right is the reverse process, where bit 7 gets the carry flag value, and bit 0 shifts into the carry flag.

Tip: When you shift left, you multiply the number by 2. Shifting right is the same as dividing by 2.

By combining shifts and addition you can perform many multiplications in an efficient way. For example, suppose you want to multiply a number by 10 (not an uncommon thing to do). One way would be to add the number to itself 10 times in a loop. While that would work, a more efficient way would be to realize that multiplying by 10 is the same as multiplying by 8 and then multiplying by 2. Since 8 and 2 are both powers of 2, you can do those multiplications using shifts.

Here is an example of both styles of multiplication:

```
device  pic16c55,oscxt5
device  turbo,stackx_optionx
reset   start_point
freq    50000000   ; 50 Mhz
```

4

```
                  org         8     ; data start
value             ds          1
result            ds          1
result2           ds          1
counter           ds          1
                  watch value,8,udec
                  watch result,8,udec
                  watch result2,8,udec
                  val = 21

                  org         0         ; code start
start_point                         ; multiply by 10 2 different ways
                  mov         value,#val
; first a loop
                  mov         counter,#10
                  clr         result
                  mov         w,value
mloop             add         result,w
                  djnz        counter,mloop
; ok answer is in result
                  nop
                  mov         value,#val
; now do shift add
                  clc
                  rl          value ; value = value *2
                  mov         result2,value
                  clc
                  rl          value
                  clc
                  rl          value       ; value = value *8
                  add         result2,value
                                          ; same answer in result2

                  sleep
```

Tip: Don't forget to clear the carry before rotating if you are using rotation for a multiply or divide. The carry bit shifts into the word which can throw off your results if you don't clear it first.

Of course, if you can't decompose your multiplication into something you can do with rotates, you'll have to look at the techniques covered in the next unit. Unfortunately, there is no easy way to combine divisions. You can divide by 2, 4, 8, or any power of two, but there isn't an easy way to divide by 10 or other arbitrary numbers.

Summary

Wow! This unit covers a lot of ground. You learned about **ADD**, **SUB**, **ADDB**, **SUBB**, lots of bit operations, and even some conditional jumps. Using these instructions you can do lots of different things including simple math, controlling the number of times a piece of code executes, and comparing numbers. These are the building blocks that allow your microcontroller to make decisions.

Remember in Unit I you read that a computer reads inputs, does processing, and generates outputs. The instructions in this chapter are the core that you will use to do the processing parts.

Exercises

1. Change the counter program to use **inc** instead of **add**. Do you still need **addb**? If you do, which bit should you add?
2. Change the counter to use a 32-bit count instead of two bytes. Test your changes using the debugger.
3. Write the program that generates a 1kHz tone on a speaker connected on pin 7 of port B. Note: don't hook a regular speaker directly to the SX output pins. Instead, use a piezoelectric speaker designed for direct IC drive. If possible, measure the output with an oscilloscope or frequency counter.

4

Answers

1. If you use **inc**, but remember that **inc** does not set the carry flag. However, it does set the zero flag. If you increment a number and get a zero, then it stands to reason that an overflow occurred. The correct code would look like this:

```
inc counter
addb counter+1,status.2  ; status.2 is zero flag
```

2. This is just a matter of changing the **ds** statement to reserve 4 bytes instead of 2 and adding two more **addb** instructions immediately following the one that is there:

```
add   counter,#1
addb  counter+1,status.0
addb  counter+2,status.0
addb  counter+3,status.0
```

3. Here is one possible solution:

```
            device      pic16c55,oscxt5
            device      turbo,stackx_optionx
            reset       start_point
            freq        50000000    ; 50 Mhz
            org         8           ; data start
delay       ds          1
delay1      ds          1
            org         0         ; code start
start_point
            mov         !rb,#$7F    ; speaker output only
loop        not         rb        ; toggle bits
            mov         delay1,#24
oloop       clr         delay
wloop       djnz        delay,wloop
            djnz        delay1,oloop
            jmp         loop
```

The programs and information in this tutorial are presented for instructional value. The programs and information have been carefully tested, but are not guaranteed for any particular purpose. The publisher and the author do not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors or omissions. The publisher and author assume no liability for damages resulting from the use of the information in this tutorial or for any infringement of the intellectual property rights of third parties that would result from the use of this information.

Rev1.2

Unit V. Advanced Flow Control

Unit V from Introduction to Assembly Language with the Scenix SX Microcontroller

© 1999 by Parallax, Inc. All Rights Reserved. By Al Williams, AWC

In the last unit, you learned how to control the flow of execution based on conditions. Instructions like **jz**, **jc**, and **djnz** allow you to jump when some condition is met. There are other ways that you can control the flow of your program, however, and you'll read about these in this unit. In addition, you'll read about ways to perform integer multiplication and division using several techniques.

5

Skipping

All of the jump instructions you read about in the last unit are not really machine language instructions. Instead, they are multi-instruction constructs that the assembler provides for your convenience. The SX actually only performs conditional tests as skips. The idea is to execute an instruction that, depending on the condition, will either execute the next instruction or skip if. It the next instruction is a **jmp** then you have an equivalent of the jump instructions you found in the last unit.

There are two things to consider here. First, the skipped instruction need not be a **jmp**. This can lead to faster, more efficient code in some cases. The second issue, however, is that skips only skip one machine language instruction. Many of the instructions you use are really assembly instructions and they consist of more than one machine language instruction (see Table V.I).

For example, some **mov** instructions require two words. Consider this bit of code:

```
skip
mov 8,#100
```

The **skip** instruction is supposed to cause the SX to skip the next instruction no matter what. However, it causes it to skip the next machine language instruction. There is no machine language instruction that corresponds to a **mov** of a constant (or literal) to a register (other than **w**). So the assembler really generates:

```
skip
mov w,#100
mov 8,w
```

The net result is that the program moves **w** – whatever happens to be in it – to register 8 without loading 100 into it first. Not what you expected. For this reason, you must be very careful when using skips.

You won't have much call to use the unconditional skip instruction. What you usually want is an instruction that skips on some condition. There are six skip instructions of this sort. The **sb** and **snb** instructions skip if a specified bit is set or clear. The assembler also provides special shorthand instructions for testing the carry (**sc** and **snc**), and the zero flag (**sz** and **snz**).

Instruction	Words
ADD (without W)	2
ADDB	2
AND (without W)	2
CJA	4
CJAE	4
CJB	4
CJBE	4
CJE	4
CJNE	4
CSA	3
CSAE	3
CSB	3
CSBE	3
CSE	3
CSNE	3
DJNZ	2
IJNZ	2
JB	2
JC	2
JNB	2
JNC	2
JNZ	2
JZ	2
LCALL	1-4
LJMP	1-4
LSET	0-3
MOV (some forms)	2
MOVB	4
OR (without W)	2
RETW (with multiple values)	varies
SUB (without W)	2
SUBB	2
XOR (without W)	2

Table V.5 - Multi-word Instructions

Comparing

Of course, a very common thing to do is to test two values and based on the result jump to some location. You saw this in the last unit done with a subtraction and a jump instruction. The assembler allows you to use special multi-instruction compares as a shorthand notation for doing this. You can find a list of these in Table V.2. These instructions require three pieces of information: a register, a register or a constant, and a jump address.

Instruction	Use	Basic Equivalent	Skip Form
CJA A,B,LBL	Jump if above	if A>B then LBL	CSA
CJAE A,B,LBL	Jump if above or equal	If A>=B then LBL	CSAE
CJB A,B,LBL	Jump if below	If A<B then LBL	CSB
CJBE A,B,LBL	Jump if below or equal	If A<=B then LBL	CSBE
CJE A,B,LBL	Jump if equal	If A=B then LBL	CSE
CJNE A,B,LBL	Jump if not equal	If A<>B then LBL	CSNE

Table V.6 - Compare Instructions

These compare instructions are very similar to a Basic or C **if** command. The only difference is that the comparison can only be between two variables or a variable and a constant. You'll find the equivalent Basic syntax in Table V.2.

You can also do a compare and skip the next instruction if the comparison is true. Just like any skip instruction, however, you have to be careful not to try to skip a multi-word instruction (see Table V.1). Table V.2 shows the corresponding skip and jump instructions.

Using Call and Return

You'll often find yourself doing the same things several times in one program. For example, if you want to add two 16-bit numbers, it is a good bet that you need to do it in more than one place.

The SX knows that you will want to write code that you can reuse and so it provides **CALL** and **RET** instructions. These instructions implement the same sort of functions that **GOSUB** provides in Basic (or functions in C).

In the last unit, there is a program that generates a 1kHz tone from a speaker connected to pin 7 of port B. But suppose you needed a program that did the following:

1. Make a 1 second beep on the speaker
2. Wait for you to push a button connected to port B, pin 0
3. Beep for 1 second again
4. Return to step #2

You can find the circuit required for this example in Figure V.1. The code in the last unit that made the 1kHz tone looks like this:

```
loop            not   rb        ; toggle bits
                mov   delay1,#24
oloop           clr   delay
wloop           djnz  delay,wloop
                djnz  delay1,oloop
                jmp loop
```

Since each loop requires about 500uS, you will need to execute the loop 2000 times to generate a 1 second tone. That simply requires another loop. However, it seems a waste to have to duplicate this code in two different parts of the program. That is where the **call** instruction is useful. You can make a subroutine out of the beep code and then call it from different parts of your program.

To create a subroutine, you simply assign the code a label. Other parts of your program will use this label (along with **call**) to execute the subroutine. When the subroutine code executes a **ret** (return) instruction, execution resumes with the instruction after the **call**. Consider the tone code rewritten as a subroutine:

```
beep            mov   second,#$DO         ; 2000 is $7D0
                mov   second+1,#$07

loop            not   rb                  ; toggle bits
                mov   delay1,#24
oloop           clr   delay
wloop           djnz  delay,wloop
                djnz  delay1,oloop
; repeat 2000 times
                djnz  second,loop
                djnz  second+1,loop
                ret                       ; go back to wherever
```

Now the main part of the code can simply use **call beep** anywhere it wants a one second beep to occur. It is perfectly acceptable to have more than one entry point into the subroutine. For example, if you wanted to set the **second** variable in your main program, you could call **loop** instead of **beep** (although you'd probably want to give it a better name). You could also get a half beep like this:

```
                mov   second,#$E8 ; 3E8
                mov   second+1,#$03
                jmp   loop
```

Subroutines can call other subroutines, but the SX can only handle 8 levels of nesting subroutines. That is, if subroutine A calls subroutine B, and subroutine B calls subroutine C, and so on, the SX will get confused when subroutine H calls subroutine I.

> Tip: This in no way limits the number of subroutines you can have in a program. It simply limits the number of subroutines you can have active at one time.

To help you understand the idea of nested subroutines and the limit on nesting, think about an elevator that can hold 8 people. Each time you execute a **call** instruction, you are putting someone else on the elevator. Each time a **ret** instruction (or a **retw** instruction; see below) executes, someone gets off the elevator. If you execute 8 **call** instructions in a row without returning, the elevator becomes full and you can't add any more people until someone gets out of the elevator. However, over the course of the day many people might ride the elevator (some more than once, even). As long as no more than 8 at a time ride, everything works.

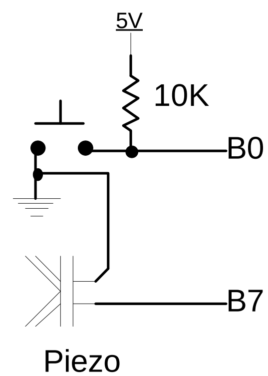

Figure V.5 – A Speaker and Switch connected to the SX

Here is the tone program:

```
            device      pic16c55,oscxt5
            device      turbo,stackx_optionx
            reset start_point
            freq        50000000   ; 50 Mhz

            org         8
second      ds          2           ; counter for 1 second tone
delay       ds          1
delay1      ds          1


            org         0
start_point mov         !rb,#$7F    ; make speaker output
            call        beep
; wait for input button
bwait       jb          rb.0,bwait
            call        beep
            jmp         bwait

; subroutine

beep        mov         second,#$d0       ; 2000 is $7D0
```

```
            mov         second+1,#$07

loop        not         rb                 ; toggle bits
            mov         delay1,#24
oloop       clr         delay
wloop       djnz        delay,wloop
            djnz        delay1,oloop
; repeat 2000 times
            djnz        second,loop
            djnz        second+1,loop
            ret                            ; go back to wherever
```

Tip: What if you wanted to use this subroutine in a program that already used labels like **oloop**, **loop**, and **wloop**? To prevent conflicts, try to use local labels (like **:oloop**, **:loop**, and **:wloop**) in your subroutines.

A few notes about this program are in order. For one thing, this is the first program in this tutorial that reads some input. The switch is connected in such a way that bit 0 of port B will read a 0 when you push the switch. The **jb** instruction tests for this – if the bit is a 1, it just loops to **bwait**.

Buttons are mechanical devices, and as such they exhibit bounce. That means that when you press the switch, the SX may see the switch open and close many times for a few microseconds until the switch firmly closes. The same thing happens when you release the switch – the button seems to turn on and off rapidly until it finally settles in the off position. In this program, this is no big deal because the tone forces a one second wait before the SX reads the switch again. However, if you were rapidly reading the button, you'd need to take this mechanical bounce into account.

If you run this program and hold the button down, the tone will continue until you release the button. That's because the program does not wait for you to release the button before continuing.

Often subroutines want to return some data (perhaps a status code) in the **w** register. To accommodate this common task, the SX provides the **retw** instruction. The **retw** instruction returns a constant in the **W** register. So:

```
      retw  #$FF
```

is the same as:

```
      mov   w,#$FF
      ret
```

Of course, **retw**, is only a single instruction so it executes faster and requires less space.

Tables

One important use of **retw** is to generate tables. Suppose you wanted to find the square of a number between 0 and 10. You know that multiplication is difficult to do, so it makes sense to simply store the values in a table and read them out instead of doing the actual calculations. Here is a subroutine that does this:

```
; square a number from 0 to 10 in the W register
; return result in the W register
square      jmp   PC+W
            retw  #0
            retw  #1
            retw  #4
            retw  #9
            retw  #16
            retw  #25
            retw  #36
            retw  #49
            retw  #64
            retw  #81
            retw  #100
```

When the main program calls the **square** routine, it jumps to a different return instruction depending on the value in **W**. The **retw** instruction loads the correct value into **W** and returns to the caller. This is simple, efficient, and very fast. It is also so common, that the assembler lets you write multiple values on the same line. So you could replace the **square** routine with two lines of assembly:

```
square      jmp   PC+W
            retw  #0, #1, #4, #9, #16, #25, #36, #49, #64, #81, #100
```

The generated machine language code is exactly the same in either case, so there is no difference in using either method. It is a matter of personal preference.

Indirection

When you access the SX's registers, you need to know the address you want to use. Early in this tutorial, you used numeric addresses (like 8 or 9), but soon you saw the advantage to using symbolic names (like **status** or **counter**). However, sometimes you don't know the exact address you want to access. For example, suppose you wanted to clear all the user memory in the SX. You could write:

```
      clr   8
      clr   9
      clr   10
      clr   11
            .
            .
            .
```

However, that seems wasteful. It would be nice if you could use a loop to index through the different registers. That is the purpose of the special **FSR** (File Select Register) and **IND** (Indirect) registers. The **IND** register is not an ordinary register. Instead it is an alias for another register somewhere in the SX. Where? Whichever address is in **FSR**.

Here is a simple example:

```
R1    EQU   10    ; register 10 is R1
R2    EQU   11    ; register 11 is R2
      mov   R1,#100
      mov   R2,#200
      mov   FSR,#10  ; store address 10 in FSR
      mov   w,IND
; W now contains 100
      inc   FSR    ; go to next address
      mov   w,IND
; W now contains 200
      mov   FSR,#R1
      mov   w,IND
; W contains 100 again
      clr   IND   ; R1 is now 0!
```

Notice that you can write to **IND** as well as read from it. **IND** is a complete alias for whatever register number you store in **FSR**.

Tip: You'll usually want to load a constant number into **FSR**. In the previous example, for instance, if you used: **mov FSR,R1** this would load the contents of **R1** (100) into **FSR** – probably not what you meant.

Here is a bit of code that will clear all the user registers in a loop:

```
            mov FSR,#8
 :loop      clr ind
            inc FSR
            jnb FSR.5,:loop
```

This takes advantage of the fact that when **FSR** reaches $20 (that is, bit 5 is set for the first time) the looping is done. You could just as easily compare **FSR** with $20 or use some other scheme to break out of the loop.

This technique is not just for clearing memory. When programming, you'll often want an array of data (for example, the last 4 samples from a sensor, or the last 8 bytes read from a serial port). Using indirection is the way to efficiently code arrays, lists, and other data structures.

Math Functions

Armed with the ability to loop and test, you can tackle arbitrary multiplication and division problems with ease. A simple-minded approach to multiply, for example, 9 by 7 is to add 9 to itself 7 times. However, with a little knowledge of binary numbers, you can write a smarter algorithm.

Remember how you learned to multiply in grade school? You'd write your problem out and multiply the results digit by digit, moving to the left with each digit. Then you'd add all the partial results up to find the correct answer. The computer can do this too. As a bonus, the SX uses binary so each partial result can only be the original number shifted to the left some number of places or 0. Think about multiplying %1001 by %101 (9 by 5).

```
          1001
X          101
----------------
        001001
        000000
+       100100
----------------
        101101 = 32 + 8 + 4 + 1 = 45
```

Performing multiplication in this fashion is known as Booth's algorithm (an algorithm is just a fancy name for a set of program steps). Here is a bit of SX code that will multiply the byte in register **V1** by the byte in register **V2**:

```
      clr V3            ; zero result
      mov ctr,#8        ; 8 bits
mloop rr V2             ; load bit 0 of V2 into carry bit
```

```
      jnc noadd         ; skip on no carry
      add V3,V1         ; add to result
noadd rl V1             ; shift V1 over 1 place
      djnz ctr,mloop    ; go 8 times
```

Of course, the result (**V3**) is a byte, so you can't multiply numbers that will require an answer larger than 255. You can easily extend this algorithm to handle more bits.

Division

You can use a similar algorithm to do division. If you remember your high-school math, dividing requires a divisor, a dividend, and produces a quotient. So when computing 20 divided by 5, 20 is the dividend and 5 is the divisor. The result, 4, is the quotient. Since 5 goes into 20 evenly, there is a remainder of 0.

When you perform division on paper, you reduce it to a series of subtractions. You also have to shift your position to keep track of what digit you are examining. The SX can do the same thing in binary. Since binary only has 1s and 0s, it is easy to tell if one number will "go into" another; simply see if the first number is smaller or equal to the second number.

Consider these program steps (or algorithm, if you prefer):

1) Set the quotient to 0
2) Shift the divisor to the left until the topmost bit is a 1
3) Remember how many shifts you performed in step 2 and add 1 to this count
4) Shift the quotient to the left (multiply by 2)
5) Compare the dividend and the divisor; if the dividend is greater than or equal to the divisor, subtract the divisor from the dividend and add 1 to the quotient
6) Shift the divisor to the right
7) Subtract 1 from the count and if not zero, return to step 4

Suppose you want to divide 20 by 5. After performing steps 1 to 3, you'll have a divisor of 160 and a count of 6. Here is the looping part of the algorithm right after performing step 6:

Dividend	Divisor	Quotient	Counter	Comments
20	160	0	6	Shifted out 5 zeros; no subtraction
20	80	0	5	No subtraction
20	40	0	4	
0	20	1	3	Subtracted
0	10	2	2	
0	5	4	1	

What about a division with a remainder? If you replace 20 in the above table with, for example, 22 you'll find that the dividend column has a 2 in it after the subtraction. Since the divisor never goes below 2, the answer is the same. However, the dividend column winds up with the remainder (2).

Here is a simple division program written for the SX:

```
            device      pic16c55,oscxt5
            device      turbo,stackx_optionx
            reset       start_point
            freq        50000000   ; 50 Mhz

            org         8
dividend    ds          1
divisor     ds          1
quotient    ds          1
counter     ds          1
            watch dividend,8,udec
            watch divisor,8,udec
            watch quotient,8,udec
            watch counter,8,udec

            org         0
start_point
            mov         dividend,#20
            mov         divisor,#5
            call        divide
            break
            nop
            sleep

; subroutine

divide      clr         counter     ; assume not dividing by zero
            clc
:loop       rl          divisor
            inc         counter
            jnc         :loop
; restore divisor so top bit is 1
            rr          divisor
; counter has number of bits in quotient
            clr         quotient
:dloop
```

```
            test        counter
            jz          :done
            clc
            rl          quotient
            cjb         dividend,divisor,:dloop1
            sub         dividend,divisor
            inc         quotient
:dloop1
            dec         counter
            clc
            rr          divisor
            jmp         :dloop
:done
            ret         ; go back to wherever
```

One thing this program does not do is test for divide by zero, which is an error. It would be simple to add a **test** instruction to set the zero flag if **divisor** was zero and jump to an error routine.

Summary

In this unit you've read about instructions that compare two values and make a decision based on the result. This type of flow control is crucial to implementing advanced multiplication and division algorithms (as well as for many other programming tasks). This unit also brought up subroutines (via the **call** and **ret** instructions) and ways to use subroutines to implement tables of constants. You can also create tables using the indirection registers (**fsr** and **ind**) that allow you to access registers without hard coding their addresses.

At this point in the tutorial, you have all the tools necessary to write some powerful programs. In the next three units you'll learn how to access all of the SX memory and how to further control the hardware. In addition, you'll work with interrupts and virtual peripherals.

Exercises

1. The example program in this unit beeps when the button is pressed for a short time. However, if the button remains depressed, the tone continues. Alter the program so that after the tone, the program waits until you release the button. Be sure to take steps to combat bounce.
2. Count the number of times the button is pressed. After 10 times, put the processor to sleep.
3. In earlier units, there is a blinker program that uses **sleep** and the watchdog timer to pause in between flashes. However, this precluded initializing the LEDs to a known state because the program could not tell the difference between the first reset and a reset after the **sleep** instruction timed out. Recall that the **status** register's bit 4 is 0 when a watchdog timeout occurs. Change the program to initialize port B to $AA in the event of a hard reset. The original program is below for your reference.

```
            device      pic16c55,oscxt5
            device      turbo,stackx_optionx, watchdog
            reset start_point
            freq  50000000   ; 50 Mhz

            org         8
pattern     ds          1

            org         0

start_point mov         !rb,#0    ; make all of port b outputs
            xor         pattern,#$FF
            mov         rb,pattern
            sleep
```


4. Connect buttons (as shown in Figure V.1) to Port B pins 0, 1, 2, and 3. Connect a piezoelectric speaker to port B pin 7. Construct a program that plays a different tone for 500mS each time you press a button. With more buttons, this could be the basis for a child's organ or a musical annunciator.

Answers

1. Here is the main code:

```
start_point    mov         !rb,#$7F    ; make speaker output
               call        beep
   ; wait for input button
bwait          jb          rb.0,bwait
               call        beep
bwait1         jnb         rb.0,bwait1
; wait for bounce to complete
               clr         delay
:dwait         djnz        delay,:dwait
               jmp         bwait
```

The delay allows time for the button to quit bouncing – the time is arbitrary and might require adjustment depending on the kind of switch you use.

2. Here is an excerpt from the solution:

```
         org   8
second         ds          2   ; counter for 1 second tone
delay          ds          1
delay1         ds          1
presses        ds          1


               org         0
start_point    mov         !rb,#$7F    ; make speaker output
               call        beep
               clr         presses
                                 ; wait for input button
bwait          jb          rb.0,bwait
               call        beep
               inc         presses
               cje         presses,#10,halt
bwait1         jnb         rb.0,bwait1
; wait for bounce to complete
               clr         delay
:dwait         djnz        delay,:dwait
               jmp         bwait

halt           sleep
```

Of course, it would be just as legitimate to store 10 in the presses variable and decrement it. This would be somewhat more efficient because you could test the zero flag after decrementing the variable, thus saving a step.

3. The solution is to simply test for the bit 4 being clear:

```
         device      pic16c55,oscxt5
         device      turbo,stackx_optionx, watchdog
         reset start_point
         freq  50000000   ; 50 Mhz

               org         8
pattern        ds          1

               org         0
```

```
start_point mov      !rb,#0    ; make all of port b outputs
; check for real reset
         jnb         status.4,agn
         mov         pattern,#$AA
agn      xor         pattern,#$FF
         mov         rb,pattern
         sleep
```

You could make an argument for setting **pattern** to $55 instead of $AA since the very next instruction will invert the bits, but either way the result is acceptable.

4. There are several ways you could complete this exercise, depending on your personal preferences. The tricky part is realizing that since each tone takes a different amount of time, you have to adjust the number of cycles to get 500mS. For example, a 1kHz tone has 500uS cycles, so you need 1000 cycles to get 500mS. However, a 2kHz tone has 250uS cycles and therefore requires 2000 cycles to maintain the same duration. Here is one solution:

```
            device      pic16c55,oscxt5
            device      turbo,stackx_optionx
            reset start_point
            freq        50000000   ; 50 Mhz
            org         8
second      ds          2   ; counter for 1 second tone
delay       ds          1
delay1      ds          1
tone        ds          1   ; tone constant
            org         0
start_point mov         !rb,#$7F    ; make speaker output
      ; wait for input button
bwait       jnb         rb.0,bp0
            jnb         rb.1,bp1
            jnb         rb.2,bp2
            jb          rb.3,bwait
; tone 3
            mov         tone,#48
            mov         second,#$01
            mov         second+1,#$01

bp          call        beep
            jmp         bwait

bp2         mov         tone,#24
```

```
             mov         second,#$FD
             mov         second+1,#$01
             jmp         bp

bp1          mov         tone,#12
             mov         second,#$FA
             mov         second+1,#$03
             jmp         bp

bp0          mov         tone,#6
             mov         second,#$F4
             mov         second+1,#$07
             jmp bp

; subroutine
beep
loop         not         rb                  ; toggle bits
             mov         delay1,tone
oloop        clr         delay
wloop        djnz        delay,wloop
             djnz        delay1,oloop
             djnz        second,loop
             djnz        second+1,loop
             ret                     ; go back to wherever
```

The programs and information in this tutorial are presented for instructional value. The programs and information have been carefully tested, but are not guaranteed for any particular purpose. The publisher and the author do not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors or omissions. The publisher and author assume no liability for damages resulting from the use of the information in this tutorial or for any infringement of the intellectual property rights of third parties that would result from the use of this information.

Rev1.2

Unit VI. Low-Level Programming

Unit VI from Introduction to Assembly Language with the Scenix SX Microcontroller

© 1999 by Parallax, Inc. All Rights Reserved. By Al Williams, AWC

In the previous units you've written programs that do simple input and output. However, the SX has many powerful I/O features that you can use if you know how they work. Besides input and output capabilities, the SX has more program and data storage than previous programs have used. To access this extra memory, you'll need to understand a special technique called banking.

Port Control

The SX has three I/O ports: ports A, B, and C. Of course, the 18-pin device doesn't have a port C, but otherwise is exactly the same as its large-package cousins. Port A only has 4 pins. Ports B and C have 8 bits each. You can read and write the pins on a port by accessing the corresponding data register (**ra**, **rb**, or **rc**). You've also seen that you can change the direction of each bit by changing the control register for the port (**!ra**, **!rb**, or **!rc**).

However, the control register gives you much more control over the ports than just the direction. Using the control register you can set other options including the threshold voltage for each pin and set if the pin uses a Schmitt trigger input or a normal logic-level input. You can also elect to turn on an optional pull up resistor on each pin.

How can a single control register have this much capability? It can't. The trick is that the control register has multiple personalities determined by the **M** or mode register. By default, the **M** register (a 4-bit register) contains $F, which makes the control registers direction registers. When you write a 0 to the control register, it makes the corresponding bit an output, and a 1 makes the bit an input.

If you set the mode register to $E, for example, the control register selects which pins have pull up resistors connected internally. Each bit that is a zero will set a pull up resistor on. Pull up resistors prevent input pins from assuming random states if there is no external circuitry driving the pin. You can set pull up resistors on any of the three ports, by setting **M** to $E and then accessing the **!ra**, **!rb**, or **!rc** registers.

You can use the **mov** instruction to load the **M** register with the contents of another register or a literal. You can also use the **mode** instruction to load a literal into **M**. Table VI.1 shows the effects of the control registers for different values of **M** (note that this table does not show settings that pertain to interrupts, a topic covered in the next unit).

Mode	!ra	!rb	!rc	SX Name
$F	Direction	Direction	Direction	TRIS_
$E	Pull up	Pull up	Pull up	PLP_
$D	Threshold	Threshold	Threshold	LVL_
$C	N/A	Schmitt	Schmitt	ST_

Table VI.1 – Mode Settings

If you set a threshold bit to 0, the SX will read the input through a CMOS-compatible buffer. This buffer will treat levels below 30% of the supply voltage (say 1.5V if the supply is 5V) as a 0. Anything above 70% of the supply voltage (3.5V) will be a 1. Voltages in between will result in an unpredictable bit, although practical experience shows that the threshold is about 50% of the supply voltage (but Scenix does not specify this).

When the threshold bit is 1, the input uses a TTL-compatible buffer. Using a TTL compatible buffer treats a 0 as .8V or less and a 1 as anything over 2V. For most modern logic circuits, this is acceptable, but interfacing with certain devices may require one setting or the other. Also, when mixing analog circuitry with the processor, you might want to adjust the thresholds to read a particular voltage level.

Ports B and C can use a Schmitt trigger input if you set a zero into the Schmitt register. A Schmitt trigger uses different thresholds depending on the situation. Imagine you are trying to set the temperature of your swimming pool to a particular temperature (say 25 degrees Celsius). You turn on your water heater, and watch the thermometer. When the temperature gets to 25, you turn the heater off. However, the pool loses heat quickly so almost immediately, the temperature drops again and you turn on the heater again. Soon you are turning the heater on and off every few seconds, never able to attain 25 degrees for more than a split second.

A Schmitt trigger uses hysteresis to battle this sort of problem. The idea is that the Schmitt trigger will use one threshold to recognize 0 to 1 transitions and another threshold to identify 1 to 0 transitions. A Schmitt trigger might see a voltage rising from .8 to .9V and output a logic 1 (5V). However, it might require that the voltage drop below .5V before returning to the zero state. This prevents a noisy or slow rising signal from causing multiple changes on the output. The SX's Schmitt triggers use 15% and 85% of the supply voltage as trip points. Once the signal rises above 85% of the supply voltage (4.25V for a 5V supply), the input reads a 1. It will continue to read a 1 until the input drops below 15% (.75V).

This can be important when dealing with inputs from real-world sensors, or noisy inputs from long lines. You can also use it to "square up" a signal – for example, reading a digital input from a charging capacitor. Of course, using the Schmitt trigger option overrides the threshold settings for the pin.

Tip: Be sure you know the state of the **M** register before you use the control registers. A common mistake is to set the **M** register to some value other than $F, use the control register, and then later try to access the control register to set direction bits. The **M** register stays at the last value you set until a reset occurs.

Analog Capabilities

The SX has one more special capability on port B. Pin 1 and 2 of port B can function as an analog comparator. You can read the comparator's output in software and you can cause pin 0 of port B to reflect the comparator's output as well.

To enable the comparator, you simply set the **M** register to 8 and write a value to **!rb**. A value of $C0 will turn the comparator function off. To turn it on, write either $40 or $00 to **!rb**. If you use $00, the comparator will operate, and pin 0 will act as a comparator output. If you use $40, pin 0 will be free for normal I/O, but the comparator will still function.

To read the state of the comparator, make sure **M** contains 8 and write to the **!rb** register. When you write to the comparator register (that is, **M** is equal to 8 and you perform a **mov** to **!rb**) the SX does a little trick behind your back. Instead of simply moving the data to the comparator register, it actually exchanges the **W** register with the comparator register. This is true even if you write:


```
mov !rb,#0
```

Because this is really the same as writing:

```
mov W,#0
mov !rb,W
```

So after writing to the comparator register, the **W** register contains the previous contents. You should only examine bit 0, the comparator status bit, after you've already enabled the comparator with another instruction. If bit 0 is high, then the voltage on B2 is higher than the voltage on B1. If it is low, then the opposite condition is true.

Why would you want a comparator input? Maybe you want the SX to compare the voltage from a potentiometer and a thermocouple. Perhaps you want to divide down your battery voltage and compare it to a known reference so you can detect when your battery is low.

Register Banking

Earlier, you read that the SX has over 100 registers. That might seem odd, because the debugger is only showing 32 registers. If you examine the SX instruction set, you'll also see that there is only room for 5 bits of data to specify a register. So how can 5 bits refer to over 100 registers? The answer is banking.

Your program does have access to 136 memory locations (not including the special registers like **ind**, **fsr**, **ra**, etc.). However, it can only work with 32 registers at one time. The first 8 registers (register 0 to 7) are the special registers and you can always access them. The registers from 8 to 15 are also always accessible – the SX doesn't use them for anything, so you can do what you want with them. This accounts for 16 registers. The other 16 (registers $10-$1F) are available for you to use as you wish. However, there are really 8 sets of these registers. Which set of 16 you are using depends on the **FSR** register.

> Tip: Don't forget that in an 18-pin SX, register 7 is available for use and always accessible. On other SX devices, this register is **rc**.

Conceptually, the SX memory map consists of 8 32-bit pages. Each page has 32 registers in it. The first 16 are always the same. The last 16 are not. Each register has its own address (and in the case of the shared registers, 8 addresses). You can see this graphically in Table VI.2.

When you want to access a register, you have several choices. First, if you are using **FSR** anyway, just put the proper address into **FSR** before using **IND**. So if you want to access the last memory location, load **FSR** with $FF. Your other option is to set the top 3 bits of **FSR** before you access memory. The values you want to use are in the column headings of Table VI.2. You can store a value in **FSR**, of course, with a **mov** instruction. However, this destroys the entire register and it also requires two machine language instructions if you are using a literal value. Since most programs will want to load literals into **FSR**, there is a **bank** instruction. This instruction loads the top 3 bits of a literal into the top 3 bits of **FSR**. This is useful because you can just name the variable you want to access. For example:

```
      org   $FF
last  ds    1

      org   0
      bank  last
      mov   last,#0
```

You might wonder why the debugger did not show you these extra pages. In the **device** statement of all the previous programs, you'll find a **pic16c55** clause. This tells the SX to only use 1 bank to simulate a different device. If you specify **sx28l** (or **sx18l** for an 18-pin device) you'll get the full set of registers and memory. The current bank of registers shows up in a bright highlight compared to the inaccessible banks in the debugging window.

	FSR=$00	FSR=$20	FSR=$40	FSR=$60	FSR=$80	FSR=$A0	FSR=$C0	FSR=$E0
$00	IND	IND	IND	IND	IND	IND	IND	IND
$01	RTCC	RTCC	RTCC	RTCC	RTCC	RTCC	RTCC	RTCC
$02	PC	PC	PC	PC	PC	PC	PC	PC
$03	STATUS	STATUS	STATUS	STATUS	STATUS	STATUS	STATUS	STATUS
$04	FSR	FSR	FSR	FSR	FSR	FSR	FSR	FSR
$05	PORTA	PORTA	PORTA	PORTA	PORTA	PORTA	PORTA	PORTA
$06	PORTB	PORTB	PORTB	PORTB	PORTB	PORTB	PORTB	PORTB
$07	PORTC	PORTC	PORTC	PORTC	PORTC	PORTC	PORTC	PORTC
$08	8 registers addressable as $08-$0F, $38-$3F, $58-$5F, $78-$7F, $98-$9F, $B8-$BF, $D8-$DF, or $F8-$FF							
$09								
$0A								
$0B								
$0C								
$0D								
$0E								
$0F								
$10	$10	$30	$50	$70	$90	$B0	$D0	$F0
$11	$11	$31	$51	$71	$91	$B1	$D1	$F1
$12	$12	$32	$52	$72	$92	$B2	$D2	$F2
$13	$13	$33	$53	$73	$93	$B3	$D3	$F3
$14	$14	$34	$54	$74	$94	$B4	$D4	$F4
$15	$15	$35	$55	$75	$95	$B5	$D5	$F5
$16	$16	$36	$56	$76	$96	$B6	$D6	$F6
$17	$17	$37	$57	$77	$97	$B7	$D7	$F7
$18	$18	$38	$58	$78	$98	$B8	$D8	$F8
$19	$19	$39	$59	$79	$99	$B9	$D9	$F9
$1A	$1A	$3A	$5A	$7A	$9A	$BA	$DA	$FA
$1B	$1B	$3B	$5B	$7B	$9B	$BB	$DB	$FB
$1C	$1C	$3C	$5C	$7C	$9C	$BC	$DC	$FC
$1D	$1D	$3D	$5D	$7D	$9D	$BD	$DD	$FD
$1E	$1E	$3E	$5E	$7E	$9E	$BE	$DE	$FE
$1F	$1F	$3F	$5F	$7F	$9F	$BF	$DF	$FF

Table VI.2 – SX Memory Map

Tip: If you organize your registers based on your usage of them, you can name your banks meaningfully. Not only does this make your code more readable, but it will often reduce the amount of switching necessary, as well. For example, suppose you have one bank of variables (bank $20) reserved for math calculations, and another for external communications (bank $40). You can define two symbols, **math** and **extcomm**, so you can write:

```
      bank math  ; switch to math bank
```

Program Pages

Another place where the SX hides extra memory is in the program space. Although none of your programs have needed it so far, the SX has 4 pages of program memory, and each page is 512 instructions (remember, instructions on the SX are not bytes). So you can use up to 2K instructions.

However, using more than 512 instructions requires careful planning. Every jump instruction (except **jmp w**, **jmp pc+w**, and **ljmp**) only take 9 bits for an address. The extra bits required come from the top 3 bits of the **status** register. Instead of manually setting these bits, however, you can force the assembler to do it for you. Just put an "@" character before the address, like this:

```
      JMP   @FarAwayPlace
```

This actually produces the following instructions:

```
      PAGE  FarAwayPlace
      JMP   FarAwayPlace
```

The **page** instruction sets the status register bits to match the target address. Since using the @ sign requires extra space, you should only use it in cases where the target address resides in a different page.

To complicate things, calling a subroutine across page boundaries is even more difficult. The **call** instruction only takes 8 bits of address. The ninth bit is set to 0, and the remaining bits come from the **status** register just like as with **jmp**. That means that a subroutine call can only occur to the first 256 instructions of a page.

This seems like a harsh restriction, but in reality, it is easy to overcome. If you can't organize your subroutines so that they are all in the first half of a page, just place a **jmp** to the subroutine (a single instruction) in the bottom half of the page, and call that instead. Don't forget that data tables (like the ones in Unit 5) are really subroutines so they have the same limitation – the **jmp** instruction that starts the table must be in the first half of the page so that other parts of the program can **call** into the table.

It is worth noting that the program counter is 11 bits long, but the **pc** register is only the bottom 8 bits. There is no way to directly read the top 3 bits. The only access you have to these bits is when they are loaded from the top 2 or 3 bits of the **status** register.

When you call a subroutine in a different page, you need the processor to restore the full 11-bit address to the program counter. It is also handy to have it set the **status** register to the caller's page so that it can make more subroutine calls on its own page. That is the purpose of the **retp** instruction. It not only restores the full address so that the caller can continue executing, but it also sets the top 3 bits of the return address into the top 3 bits of the **status** register.

Tip: The **ret** instruction and the **retp** instruction take the same amount of space and execute at the same speed. If there is any chance you might call a subroutine from across page boundaries, use **retp**. The only exception would be if you wanted the subroutine to modify the top bits of **status**.

Reading Program Storage

In the last unit you saw how to use **retw** to form tables in program memory. There is another way you can access program memory – the **iread** instruction. This instruction takes 4 cycles (unusual for an instruction that doesn't jump or skip). It takes the **M** register and the **W** register as an 11-bit address, reads the 12-bit word at that address, and loads it into the **M** and **W** registers.

How do you get arbitrary data into the program memory? Use **DW** as in:

```
            org         0
start_point mov m,#SomeData>>8  ; top part of address
            mov w,#SomeData&$FF ; bottom part of address
            iread
            nop
            nop
            break
            nop
            sleep


SomeData    dw $1A5
```

If you debug this program, the **W** register will contain $A5 and the **M** register will contain $1 at the breakpoint.

Tip: Be careful if you access the port control registers after executing **iread** since the **M** register will not contain what you expect and that alters the control register's function.

Summary

The techniques in this unit are not that useful for the simple programs you've written up to this point. But in real life, 24 bytes of data storage and 512 instructions only go so far. The key to success with large programs is to carefully plan and organize. If you can keep related variables in the same bank, you'll be much happier.

Variables that you use in many parts of your program should be below $10 (the shared area). Of course, with only 8 bytes shared between banks (9 on an SX18), you have to be very frugal.

Organization for code is important too. Related routines on the same page do not need long jumps. You also need to be mindful of placing subroutines in the second half of any bank, since you won't be able to call them there.

If it seems odd that the SX has all these odd ways to access memory, remember that it is all in the name of compatibility. The SX is backward compatible with other processors that do not have so much memory. The price of having extra resources is extra complexity.

Exercises

1. Write a program to clear all 8 register banks. Be careful not to clear the first 8 registers (which are the special function registers like **pc** and **ind**). Also, don't clear the shared bank more than once. Can you make the clear loop a subroutine?
2. Use **org $200** to place the clearing subroutine in the above program in the first program bank. Single step through the execution.
3. Write a program to convert Celsius temperature to Fahrenheit, using a lookup table accessed with **iread**. Assume the input ranges from 0 to 29 degrees. The formula for conversion, by the way, is F=1.8C+32.

6

Answers

1. Here is one possible solution:

```
            device  sx28l,oscxt5
            device  turbo,stackx_optionx
            reset start_point
            freq  50000000   ; 50 Mhz

            org         0
start_point mov         fsr,#8     ; shared bank
            call        clear
            mov         fsr,#$10
zloop call  clear
            add         fsr,#$11
            jnc         zloop
            nop
            break
            nop
            sleep

; subroutine clears from FSR until FSR AND $F is 0
clear clr   ind
            inc         fsr
            mov         w,#$F
            and         w,fsr
            jnz         clear
            dec         fsr     ; back up
            ret
```

2. Moving the subroutine requires you to: 1) place **org $200** in front of the clear routine; 2) change each call to **clear** with one to **@clear**; and 3) change the **ret** instruction to a **retp**. Try performing each of these steps in sequence and debugging the code before making the next change.

3. Here is a simple implementation:

```
            org         8
tempm       ds          1   ; place to hold M
value       ds          1   ; value to convert


            org         0
start_point mov         value,#11 ; 11 degrees C
            call        @convert
            nop
            break
            nop
            sleep
convert     mov         tempm,m
            mov         m,#table>>8
            mov         w,#table & $FF
            add         w,value
            iread
            ; don't need M
            mov         value,w
            mov         m,tempm  ; restore M
            ret

table       dw 32,34,36,37,39,41  ; 0-5
            dw 43,45,46,48,50     ; 6-10
            dw 52,54,55,57,59     ; 11-15
            dw 61,63,64,66,68     ; 16-20
            dw 70,72,73,75,77     ; 21-25
            dw 79,81,82,84          ; 26-29
```

The programs and information in this tutorial are presented for instructional value. The programs and information have been carefully tested, but are not guaranteed for any particular purpose. The publisher and the author do not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors or omissions. The publisher and author assume no liability for damages resulting from the use of the information in this tutorial or for any infringement of the intellectual property rights of third parties that would result from the use of this information.

Rev1.2

Unit VII. Interrupts

Unit VII from Introduction to Assembly Language with the Scenix SX Microcontroller

© 1999 by Parallax, Inc. All Rights Reserved. By Al Williams, AWC

One of the great strengths of modern computers is that they can do more than one thing at a time, right? With a Windows PC, you can surf the Web, work on an e-mail, and touch up a photo from your digital camera, all at the same time. This sounds great except for one thing: most computers (including your Windows PC) only do one thing at a time.

How is this possible? While it is true that most computers can only do one thing at a time, they can do one thing very rapidly. Modern operating systems allocate small chunks of time to each active task. In this way, each task appears to run at the same time. Also, modern computers can respond to external events – for example, a keystroke or a mouse movement. This also helps with the illusion that the computer is performing many tasks since the computer can handle events as they occur instead of waiting for them.

7

To get this sort of capability, a computer needs a way to track time and it also needs a way to stop what it is doing in favor of another task. The SX has two features that work together in this area: the real time clock counter (**RTCC** register) and interrupts. The **RTCC** register does just what its name implies: it increments on a precise predetermined interval regardless of what else the processor is doing. It can also increment in response to an external pulse input. Interrupts allow an external event or a time period to trigger a piece of your program. Whatever the SX was doing before the event is put on hold until the event code (an interrupt service routine or isr) completes.

In assembly language programming, interrupts have a reputation as being difficult to use. It is true that interrupts require careful planning. However, the SX has several features that make dealing with interrupts less troublesome than with many other similar processors.

What constitutes an event? One common event is when the **RTCC** register rolls over (that is, changes from $FF to $00). You can also configure interrupts to occur on rising or falling edges on any (or all) port B pins. To use interrupts, you must configure them first – by default no interrupts occur.

The Real Time Clock Counter

One of the most common sources of interrupts is when the **RTCC** register's value changes from $FF to $00. This indicates that 256 time periods have elapsed or 256 external events occurred. Using this interrupt, you can receive interrupts at a regular time interval which is useful for keeping time, measuring pulse widths, generating pulses, and other time-sensitive operations.

What causes the **RTCC** register to increment depends on bit 5 of the **!option** register (**RTS**). If this bit is 0, the counter increases with each instruction cycle. If the bit is 1, then **RTCC** increments each time it detects a pulse on the **RTCC** pin. By using the **RTE** bit (bit 4 of **!option**) you can determine if the counter responds to rising edges (0) or falling edges (1).

By default, the **RTCC** register increments on each instruction cycle or external event. At 50MHz, then, the **RTCC** requires 20nS * 256 = 5.12uS to roll over when counting instruction cycles. This time is too short for most purposes (as you'll see shortly), so you'll often want to divide the clock cycle by some factor. You can do this by assigning the prescaler to **RTCC**. This is the same prescaler the watchdog timer uses, so you have to assign it to one use or the other. You can't scale the **RTCC** count and the watchdog timer at the same time.

To assign the prescaler to **RTCC**, clear bit 3 of the **!option** register (**PSA**). The last 3 bits in the **!option** register determine the division rate (see Table VII.1). The maximum ratio is 1:256 which at 50MHz works out to 1.3mS (.0013S). Of course, if you are using a different clock frequency all of these times will be different as well. Obviously, if you are using an external source to drive the **RTCC** pin, the time between rollovers depends on the external source.

Tip: Notice that the table does not contain a 1:1 setting. That is because a 1:1 setting is what you get when the prescaler is working for the watchdog timer.

PS2	PS1	PS0	Ratio	Roll overTime at 50MHz
0	0	0	1:2	10.24uS
0	0	1	1:4	20.48uS
0	1	0	1:8	40.96uS
0	1	1	1:16	81.92uS
1	0	0	1:32	163.84uS
1	0	1	1:64	327.67uS
1	1	0	1:128	655.35uS
1	1	1	1:256	1310.72uS (1.31 mS)

Table VII.1 – Prescaler Settings

RTCC Delays

Even without interrupts, the **RTCC** register can be useful. In previous units, programs used a programmed delay to pause for a particular interval. If the **RTCC** is incrementing with the instruction clock, you can use it to time your delays easily. Take a look at this subroutine:

```
; assume prescaler is 1:256
delay1_3ms        mov         rtcc,#1
; testing for zero is ok because the 256 prescaler is on
:wait             mov         w,rtcc
                  jnz         :wait
                  ret
```

The subroutine sets **rtcc** to 1 (which also, incidentally, clears the prescaler). It then waits for **rtcc** to equal zero. This will require 255 counts and each count requires 256 instruction cycles. Therefore, at 50MHz, the total delay is 256*255*20nS = 1.3mS.

Don't forget that writing to **rtcc** clears the prescaler. This can lead to subtle side effects. For example, you might be tempted to use the **test** instruction to test the prescaler for a zero value. This won't work because using **test** is the same as moving a register into itself. While this does test for zero, it also clears the prescaler so that the **rtcc** register never increments.

Another pitfall is testing for equality. If the prescaler is not set, **rtcc** increments on each instruction cycle. Then it would be dangerous to test for a single value of the prescaler. Why? Because **rtcc** might assume that value while you are executing another instruction. For example, suppose the subroutine above loads **w** with $FF at the **:wait** label. With prescaling off, the next time through the loop the counter will be 3 – it was zero during the **jnz** instruction!

RTCC Interrupts

To enable **RTCC** rollover interrupts, clear the **RTI** bit (bit 6) in the **!option** register. Once this bit is clear, the processor will stop whatever it is doing when **RTCC** rolls over and execute the code starting at location 0. Of course, up until now, your program started at location 0, but that is only because the **reset** directive pointed there. You can start your program further up in memory to allow for interrupt processing.

When an interrupt occurs, the SX disables further interrupts. It also saves **status**, **fsr**, and **w**. The SX then clears the top 3 bits of the **status** register (these bits form the top portion of jump addresses) and jumps to address 0. All of this work is necessary so that the interrupt service routine (ISR) does not interfere with the execution of the main program. Once the ISR is finished, it uses the **reti** instruction to restore control to the main program. This also enables future interrupts.

Tip: Unlike many other processors, the SX stores its context (the **w**, **fsr**, and **status** registers) in special temporary areas, not the stack. However, the chip does not service interrupts if they occur while still processing a previous interrupt.

Perhaps the simplest way to use the **rtcc** interrupt is to simulate a wider real time clock. Remember that even with the maximum prescaling in effect, **rtcc** rolls over every 1.3mS or so (at 50MHz). What if you wanted to delay 100mS? Sure you could call the 1.3mS delay nearly 100 times. But if you had a 16-bit **rtcc** register you could simply wait for the count to exceed 19531 (each count is worth about 5uS when the prescaler is at 1:256).

Here is a simple 100mS LED flasher based on these ideas:

```
            device      turbo,stackx_optionx
            reset start_point
            freq  50000000   ; 50 Mhz
            org   8
rtcc1       ds    1

            org         0
isr         inc         rtcc1       ; interrupt handler
            reti

start_point
            mov         !rb,#$80    ; 7 outputs, 1 input
; set RTCC to internal clock 1:256 ratio
            mov         !option,#$87
loop        xor         rb,#$FF
            call        delay100ms
            jmp         loop


delay100ms  clr         rtcc
            clr         rtcc1
:wait       mov         w,#$4c   ; $4c4b is 19531
            mov         w,rtcc1-w
            jnz         :wait
:wait0      mov         w,#$4b
            mov         w,rtcc-w
            jnz         :wait0
            ret
```

Periodic Interrupts

In the previous examples, the main program blinks an LED and controls the delay between flashes of the lamp. However, the real power to interrupts is when you allow the ISR to perform a task, seemingly while the main routine is executing. Look at this program:

```
            device      turbo,stackx_optionx
            reset start_point
            freq        50000000   ; 50 Mhz
            org         8
rtcc1       ds          1

            org         0
isr         inc         rtcc1
            cjne        rtcc1,#$4D,iout  ; blink every $4D00 periods
            xor         rb,#$FF
; reset time
            clr         rtcc
            clr         rtcc1
iout
            reti

start_point
            mov         !rb,#$80    ; 7 outputs
; set RTCC to internal clock 1:256 ratio
            mov         !option,#$87
loop
            jmp         loop
```

The main program sets **!rb**, **!option**, and then does a simple **jmp** instruction to loop forever doing nothing. All the work occurs in the ISR. It is interesting to note that the ISR resets the **rtcc** register so that the interrupt will occur periodically. This isn't unusual when you want the interrupt to repeat at a regular interval. Of course, the interval will be a little longer than you

There is one problem with this, however. A complex ISR may take a different amount of time to execute depending on the current situation. This can lead to timing errors intolerable in precise applications. For example, in the above piece of code, the **reti** instruction adds a slight delay to the total time although for this application it is negligible.

A better answer is to use the **retiw** instruction to end the ISR – especially if the prescaler is off. This instruction adds the **w** register's contents to **rtcc**. Say the processor is set so that **rtcc** will cause an interrupt when it rolls over and that the prescaler is assigned to the watchdog timer. Each count of the **rtcc** represents 20nS (assuming, as always, a 50MHz clock). When the interrupt begins the **rtcc** has already counted to 3. As the ISR

7

continues, the **rtcc** continues to increase. To accurately set the time, you have to take into consideration how much time has already elapsed. Luckily, there is a simple answer – the **rtcc** register already has this information! If you subtract the number of cycles you want between each interrupt from the number of cycles already elapsed, you are left with the exact number of cycles required.

For example, say you want an interrupt to occur every 50 cycles (1uS). You can simply use the following two lines of code at the end of your ISR:

```
mov w,#-50
retiw
```

The only catch is that your ISR, including the 3 cycle interrupt latency, must not exceed 46 cycles. If it does, you'll either miss the next interrupt, or you will return to the main program only to have an interrupt occur immediately. Because of the interrupt latency you must always allow 3 cycles plus at least enough time for one instruction to execute in the main program – figure a total of 6 cycles. However, even then your main program will not execute very often – you should allow a more generous time slice between interrupts in most cases.

A Clock Example

A computer that knows what time it is can be very useful. You might want to count down a model rocket launch, or time stamp readings from a sensor. With an accurate interrupt it is easy to keep the time. The hard part, is translating the rapid stream of interrupts into numbers more meaningful to humans. Here is a simple program that uses a 50MHz clock to the **rtcc** register. The ISR adds –50 to **rtcc** so that it generates a periodic interrupt every 1uS. The ISR maintains two 16-bit counters to count microseconds and milliseconds.

Of course, every 1000 milliseconds constitutes a second, every 60 seconds is a minute, and 60 minutes make an hour. You could easily extend this to track days if you wanted to do so. The main program in this case doesn't do anything, but you could easily add whatever code you wanted.

This is a hard program to debug because single stepping it doesn't show the correct time. You can run the program at full speed in the debugger and press the Poll button to see the time change. You'll also see LEDs on port B blink and, if you connect a piezo speaker to one of the port B pins, you'll hear your SX clock ticking.

```
            device      turbo,stackx_optionx
            reset start_point
            freq        50000000   ; 50 Mhz
            org         8
microlow    ds          1
microhi     ds          1
millilow    ds          1
millihi     ds          1
seconds     ds          1
minutes     ds          1
```

```
hours       ds    1
            watch hours,8,udec
            watch minutes,8,udec
            watch seconds,8,udec

            org         0
isr         inc         microlow
            snz
            inc         microhi
            cjne        microhi,#$03,iout  ; blink every $03e8 periods
            cjne        microlow,#$e8,iout
; 1000 uS already!
            clr         microlow
            clr         microhi
            inc         millilow
            snz
            inc         millihi
            cjne        millihi,#$03,iout
            cjne        millilow,#$e8,iout
; 1000 ms!
            clr         millihi
            clr         millilow
            xor         rb,#$FF  ; toggle LEDs
            inc         seconds
            cjne        seconds,#60,iout
; seconds roll over
            clr         seconds
            inc         minutes
            cjne        minutes,#60,iout
; minutes roll over
            clr         minutes
            inc         hours
            cjne        hours,#24,iout
; hour roll over
            clr         hours
; could track days if we wanted to

; reset time
iout
            mov         w,#-50   ; interrupt every 1uS
            retiw
```

7

```
start_point
            mov         !rb,#$00    ; all outputs
            clr         microhi
            clr         microlow
            clr         seconds
            clr         hours
            clr         minutes
; set RTCC to internal clock 1:1 ratio
            mov         !option,#$88  ; no prescale
loop
            jmp         loop
```

External Interrupts via RTCC

When you think of using the **RTCC** pin to monitor external events, you usually think of counting pulses. You can certainly do this, of course. When you set bit 4 of **!option** (the **RTS** bit), the pin monitors pulses and uses them to increment **RTCC**. If the **RTE** bit (bit 4 of **!option**) is clear, the count occurs on rising edges, otherwise the SX detects falling edges. The prescaler is still available, so you can divide the input down if you like.

However, what if you want a single external interrupt? At first glance, it would seem that you can't do this with **RTCC**. After all, even with the prescaler assigned to the watchdog timer, you still need 256 pulses to get a single interrupt, right?

While that seems true, there is a trick you can use to make **RTCC** simulate an external interrupt. Simply load the **RTCC** register with $FF. Assuming the prescaler is off and the **RTS** bit is set, the next input pulse will cause an interrupt. A simple but effective technique. Of course, the ISR will then reset **RTCC** to $FF before issuing a **reti** instruction so the interrupt will be "armed" for the next event.

Port B Multi Input Wakeup

In addition to the **RTCC** trick, you can configure any (or all) of port B's pins as external interrupts. Port B has two special registers that allow it to detect input edges. These are in effect at all times, not just when interrupts are enabled. Like other special port registers, you access these by using **!rb** while the **M** register is set to a special value. If **M** is $A, you can select which edge each pin monitors. A 1 bit in this register makes the SX detect falling edges (that is, 1 to 0 transitions) on the corresponding pin. A 0 bit detects 0 to 1 transitions or rising edges. When the selected edge appears on a pin, the SX sets the corresponding bit in the multi-input wake up (MIWU) pending register (**!rb** with **M** = $9). The SX never clears this register. When your program writes the **W** register into **!rb** and **M** is $9, the SX actually swaps the two values. So you can read the pending bits and clear them at the same time.

This processing occurs at all times. Most programs just ignore this feature. However, you can use it to detect when an edge occurred even when you aren't using the port B interrupts. If you connect the circuit in Figure VII.1 to several port B pins, you can try this program:

```
            device      turbo,stackx_optionx
            reset start_point
            freq        50000000   ; 50 Mhz
            org         8
microlow    ds          1
microhi     ds          1
millilow    ds          1
millihi     ds          1
seconds     ds          1
minutes     ds          1
hours       ds          1
edges       ds          1

            watch hours,8,udec
            watch minutes,8,udec
            watch seconds,8,udec

            org         0
isr         inc         microlow
            snz
            inc         microhi
            cjne        microhi,#$03,iout  ; blink every $03e8 periods
            cjne        microlow,#$e8,iout
; 1000 uS already!
            clr         microlow
            clr         microhi
            inc         millilow
            snz
            inc         millihi
            cjne        millihi,#$03,iout
            cjne        millilow,#$e8,iout
; 1000 ms!
            clr         millihi
            clr         millilow
            inc         seconds
            cjne        seconds,#60,iout
; seconds roll over
            clr         seconds
            inc         minutes
            cjne        minutes,#60,iout
; minutes roll over
            clr         minutes
```

```
            inc         hours
            cjne        hours,#24,iout
; hour roll over
            clr         hours
; could track days if we wanted to

; reset time
iout
            mov         w,#-50   ; interrupt every 1uS
            retiw

start_point
            mov     !rb,#$FF
areset      clr         microhi
            clr         microlow
            clr         seconds
            clr         hours
            clr         minutes
; set RTCC to internal clock 1:1 ratio
            mov         !option,#$88  ; no prescale

; Turn on port B pull up resistors
            mode        $E
            mov         !rb,#$00
; set port B pin 0 to interrupt on falling edge
            mode  $     A  ; select edge
            mov         !rb,#$FF
            mode        $9  ; enable interrupts
            mov         !rb,#%0  ; clear pending
; wait for 10 seconds
wait10      cjne        seconds,#10,wait10
            mov         !rb,#%0  ; read pending and clear
            mov         edges,w
; important: reset mode register
            mode        $F
            mov         !rb,#0   ; set to outputs
; flip sense of edge bits
            not         edges
            mov         rb,edges

loop
; active wait so ticking will occur
```

```
        jmp     loop
```

This is more or less the same program as before, but it doesn't produce the blinking lights and ticking effect. Instead, it waits 10 seconds (easy to do with the clock interrupt routine) and then turns on lights that correspond to the buttons you pushed during that 10 seconds. This is trivially easy using the MIWU feature. Since the LEDs turn on when the port outputs a 0, the program uses the **not** instruction to invert the pending bits.

Tip: This program initially sets the direction register so that all port B pins are inputs. Then, after the pause, it sets all pins to outputs. An easy mistake to make here is to forget to set the **M** register back to $F before switching to outputs. The edge detection code changes the **M** register, and you must change it back to $F before accessing the direction register.

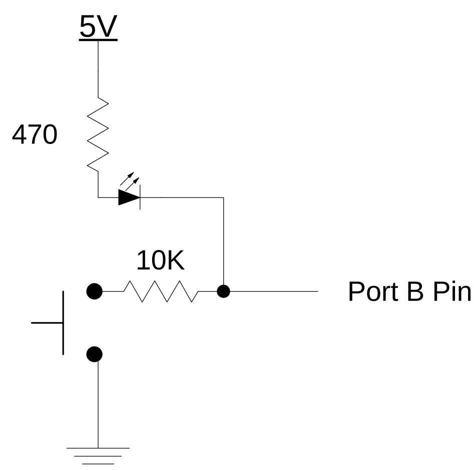

Figure VII.1 – Switch/LED Circuit

Port B Interrupts

When the SX detects an edge, it can also generate an interrupt. You can set this by clearing bits in the **!rb** register while **M** is equal to $B. When the SX detects an edge on the corresponding pin, it will generate an interrupt. It is up to the ISR to examine the pending register and clear it for further interrupts. This interrupt is exactly like an **rtcc** interrupt – it saves the SX context and starts at location 0.

It is possible to use port B interrupts and **rtcc** interrupts at the same time, but it can be tricky. For example, if a pulse occurs while the ISR executes, the SX will not generate interrupts after the ISR returns until a new event occurs. By the same token, if **rtcc** rolls over while the SX is processing a port B interrupt, you will miss the **rtcc** interrupt. In some cases, timing is not that critical, so losing a microsecond or two isn't that important. However, if you require solid time accuracy you should consider only dealing with one interrupt source (port B or **rtcc**) in one program.

Tip: If you need a real-time clock and edge detection, think about using the **rtcc** interrupt at a fast rate and simply examine the pending bits on each timer tick (this is often known as polling). For many applications, scanning the inputs every microsecond is good enough.

It is also possible to use the port B interrupt to wake up after a **sleep** instruction. If a port B interrupt occurs after a **sleep** instruction, an interrupt does not occur. Instead, the processor resets with bit 3 of the **status** register clear and bit 4 set. Although port B interrupts will interrupt the SX's sleep, an **rtcc** interrupt will not.

Summary

Interrupts need not be difficult to use. This is especially true of the SX because the chip takes care of many details for you. Interrupts are essential when you need to process inputs while doing something else, keep track of time, or generate precise outputs while doing other tasks.

Interrupts, coupled with the SX's high speed, form the basis for the virtual peripheral strategy discussed in the next unit. Although interrupt handling requires a bit of careful design, and can be difficult to debug, they are well worth the price.

Exercises

1. Write a program that uses a timer interrupt to track (at least) seconds. Normally, the program does nothing. However, when you press a button connected to pin 0 of port B, the program should flash an LED (or click a piezo speaker) every second until you push the button again. Pushing the button a third time should resume LED flashing and so on. Use the **rtcc** interrupt for timing and poll the switch in the main program.
2. Modify the above program so that the ISR samples the input switch using the MIWU capability but do not use the port B interrupts.
3. Modify the program again so that you use both interrupts; the **rtcc** and the port B interrupt.
4. Which of the three programs do you think uses the best approach?

Answers

1. The solution is straightforward. Notice you can't use the **sleep** instruction or else the program will just halt.

```
            device      turbo,stackx_optionx
            reset start_point
            freq  50000000   ; 50 Mhz
            org         8
microlow    ds          1
microhi     ds          1
millilow    ds          1
millihi     ds          1
seconds     ds          1
ticker      ds          1
tmp         ds          1


            org         0
isr         inc         microlow
            snz
            inc         microhi
            cjne        microhi,#$03,iout  ; blink every $03e8 periods
            cjne        microlow,#$e8,iout
; 1000 uS already!
            clr         microlow
            clr         microhi
            inc         millilow
            snz
            inc         millihi
            cjne        millihi,#$03,iout
            cjne        millilow,#$e8,iout
; 1000 ms!
            clr         millihi
            clr         millilow
            test        ticker
            jz          notick
            xor         rb,#$FF  ; toggle LEDs
notick      inc         seconds
iout
            mov         w,#-50   ; interrupt every 1uS
            retiw
```

```
start_point
            mov         !rb,#$01    ; 7 outputs, 1 in
areset      clr         microhi
            clr         microlow
            clr         seconds
            clr         ticker
; set RTCC to internal clock 1:1 ratio
            mov         !option,#$88  ; no prescale

loop
; active wait so ticking will occur
            jb          rb.0,loop
; button pushed
            not         ticker
; debounce delay (about 1 second)
milloop0    test        millihi   ; wait for millhi to go to 0
            jnz         milloop0
milloop1    test        millihi
            jz          milloop1  ; wait for nonzero
milloop     test        millihi
            jz          milloop   ; wait for zero again
            jmp         loop
```

2. Compared to the last program, this one has a similar ISR, but a very different main program (all the work is in the ISR). Notice that the ISR changes the **M** register, so it has to save and restore it to ensure the main program's **M** register does not change (of course, in this case, the main program doesn't care, but that will not usually be the case). To protect against bounce, the code examines the edge pending register every 1mS.

```
            device      turbo,stackx_optionx
            reset start_point
            freq        50000000   ; 50 Mhz
            org         8
microlow    ds          1
microhi     ds          1
millilow    ds          1
millihi     ds          1
seconds     ds          1
ticker      ds          1
tmp         ds          1


            org         0
isr
            inc         microlow
            snz
            inc         microhi
            cjne        microhi,#$03,iout  ; blink every $03e8 periods
            cjne        microlow,#$e8,iout
; 1000 uS already!
            clr         microlow
            clr         microhi

; check for key every 1ms
            mov         tmp,M   ; save M register
            mode        $9
            clr         w
            mov         !rb,w ; exchange w and pending
            and         w,#1  ; test low bit
            sz
            not         ticker      ; invert ticker
            mov         M,tmp   ; restore M

; roll millisecond
```

7

```
            inc         millilow
            snz
            inc         millihi
            cjne        millihi,#$03,iout
            cjne        millilow,#$e8,iout
; 1000 ms!
            clr         millihi
            clr         millilow
            test        ticker
            jz          notick
            xor         rb,#$FF  ; toggle LEDs
notick      inc         seconds
iout
            mov         w,#-50   ; interrupt every 1uS
            retiw

start_point
            mov         !rb,#$01    ; 7 outputs
areset      clr         microhi
            clr         microlow
            clr         seconds
            clr         ticker
; set RTCC to internal clock 1:1 ratio
            mov         !option,#$88  ; no prescale
; set port B detect falling edge
            mode        $A  ; select edge
            mov         !rb,#$FF

loop
            jmp         loop
```

3. This version is perhaps the least satisfactory of the three. It requires switches that don't bounce much since it is difficult to filter multiple interrupts caused by bouncing. Also, if an **rtcc** event occurs during processing for a switch closure, the time becomes inaccurate.

```
            device      turbo,stackx_optionx
            reset start_point
            freq  50000000   ; 50 Mhz
            org   8
microlow    ds    1
microhi     ds    1
```

```
millilow    ds          1
millihi     ds          1
seconds     ds          1
ticker      ds          1
tmp         ds          1


            org         0
isr
; check for pending key
            mov         tmp,M   ; save M register
            mode        $9
            clr         w
            mov         !rb,w ; exchange w and pending
            and         w,#1  ; test low bit
            jz          rtccisr
            not         ticker      ; invert ticker
            mov         M,tmp   ; restore M
            iret

rtccisr
            mov         M,tmp
            inc         microlow
            snz
            inc         microhi
            cjne        microhi,#$03,iout  ; blink every $03e8 periods
            cjne        microlow,#$e8,iout
; 1000 uS already!
            clr         microlow
            clr         microhi


; roll millisecond

            inc         millilow
            snz
            inc         millihi
            cjne        millihi,#$03,iout
            cjne        millilow,#$e8,iout
; 1000 ms!
            clr         millihi
            clr         millilow
```

```
            test        ticker
            jz          notick
            xor         rb,#$FF  ; toggle LEDs
notick      inc         seconds
iout
            mov         w,#-50   ; interrupt every 1uS
            retiw
start_point
            mov         !rb,#$01    ; 7 outputs
areset      clr         microhi
            clr         microlow
            clr         seconds
            clr         ticker
; set RTCC to internal clock 1:1 ratio
            mov         !option,#$88  ; no prescale
; set port B detect falling edge
            mode        $A  ; select edge
            mov         !rb,#$FF
            mode        $B  ; enable interrupt on pin 0
            mov         !rb,#$FE

loop
            jmp         loop
```

4. It is fairly clear that program #3 would require a great deal of work to make it robust. Mixing two interrupt sources is a risky business. Of the other two techniques, it boils down to personal taste. The code in #1 has more portions of the program in the main loop where they will be easier to debug. However, #2 is quite clean and keeps the processing out of the way of the main program (presumably, you'd be doing something in the main program).

The programs and information in this tutorial are presented for instructional value. The programs and information have been carefully tested, but are not guaranteed for any particular purpose. The publisher and the author do not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors or omissions. The publisher and author assume no liability for damages resulting from the use of the information in this tutorial or for any infringement of the intellectual property rights of third parties that would result from the use of this information.

Rev1.2

Unit VIII. Virtual Peripherals

Unit VIII from Introduction to Assembly Language with the Scenix SX Microcontroller

© 1999 by Parallax, Inc. All Rights Reserved. By Al Williams, AWC

Most if not all microcontrollers are valuable because they communicate with the outside world in some way. As a result, system designers spend a lot of time interfacing microcontrollers to the outside world. With old-fashioned processors, everything required additional electronic components. Want to read a voltage? Get an A/D (analog to digital) chip. What to talk to a PC? Get a UART (Universal Asynchronous Receiver and Transmitter) chip.

In recent years, microcontroller manufacturers have been integrating common peripheral chips directly into the microcontroller. This allows for simpler system design and conserves the controller's I/O capacity. The only problem is, no microcontroller can have every possible peripheral. For one project you might need a UART. The next project might require two A/D inputs. Still another project might require a single A/D but two UARTs. Obviously, no matter how clever the microcontroller designers are, you will never be able to have all peripherals built into the microcontroller.

Another problem with this approach is that you have to have different microcontrollers for different tasks. You can't take a microcontroller with a built-in A/D and use it in place of one that has a UART. This makes it complicated to control your inventory of microcontrollers. Ideally, you'd like to use the same part in all of your designs. At the least, you want the fewest number of different parts possible.

Scenix address this problem via Virtual Peripherals or VPs. VPs take advantage of the SX's raw speed and interrupt capability to simulate traditional peripheral devices in software instead of hardware. This has many advantages:

1) Use one part for all designs
2) Add whatever devices you need for a particular project
3) Modify devices to meet your needs – not usually possible in hardware

A VP is simply a code module (usually an interrupt service routine or ISR) that simulates an I/O device. You can download many VPs from Scenix's Web site (www.scenix.com). Other VPs may be available (for free or for a fee) from third parties. You can even write your own VPs for use in later projects or to sell to other programmers. Some VPs do require a few external components (usually a few resistors or capacitors). Others work completely in software.

Using a VP

When you begin designing a project around the SX, you should first see if there are any standard VPs that would be of use to you. Scenix releases new VPs frequently, but here are few of the more useful VPs that are available now:

- DTMF Generation – Generates TouchTones
- FSK Detection – Receives frequency shift keying data

- FSK Generation – Generates frequency shift keying
- I2C – Interface with IIC-bus chips (one VP for slave, another for master)
- SPI – Interface with SPI-bus chips (one VP for slave, another for master)
- UART – Serial I/O (up to 230.4 Kbaud)
- Multi UART – 8 serial ports each running at 19.2 Kbaud
- LCD – Drives a standard Hitachi LCD module (one VP for 4 bit, another for 8 bit)
- LED – Drives seven segment LEDs
- PWM – A variety of VPs allow you to generate pulse width modulation, useful for generating voltages, controlling motor speeds and similar tasks
- ADC – You can actually use a few common parts to make an ADC almost completely in software
- Stepper Motor – Control stepper motors
- Timers – Common VPs can implement timers and real-time clocks
- Input – VPs exist that can debounce buttons and scan keypads

Tip: Be sure to check out the latest list at http://www.scenix.com/virtual/vp/sx_library_5.pdf.

Once you select a VP, you need to integrate it into your program. You might be tempted to use more than one VP. You can do this (see below), but for now just pick one. As an example, suppose you wanted to build a circuit that would dial the Parallax telephone number using TouchTones over a piezo speaker connected to Port C pin 6.

If you look on the Scenix Web site, you'll see that there is a document file that describes the DTMF generation VP and source code to an example program. One problem is that the example program invariably does things you'd rather not do, so you have to cut and paste the pieces you want into your program.

The example program reads data from an RS-232 port and dials the number as instructed. For this example, you don't need the serial I/O VP. However, a quick examination of the example's ISR shows that it also contains PWM and timer VPs. Detailed examination reveals that both are necessary for the DTMF VP.

In addition to the ISR, you also have to get the variables that the routines use and several subroutines that help you access the VP's functions. The VP may also require specific initialization of port control registers, the **!option** register, or internal variables. In the end you may have to resort to a bit of trial and error unless you are prepared to fully comprehend what the program is doing.

Once you think you have everything you need, you might want to use the Run | Assemble command to see if you get any assembly errors. If you don't, then you probably have everything you need (although you may have extra things too if you are not careful).

Often, the VP does not use the same port assignments as you'd like to use. Usually you can interchange the pin numbers with no ill effects. However, be careful. If the VP is using, for example, port B's interrupt capabilities, you won't be able to move pins to port A or C which do not have interrupts. Usually the VP will have an equate near the top that sets the I/O definitions (**PWM_pin**, in this case). This is misleading, however. In addition to changing the equate, you also have to find all the places where the VP references the **ra**, **rb**, **rc**, **!ra**, **!rb**, or **!rc** registers and correct these lines as well.

With the VP in place, the main program is trivially simple:

```
; load digits
            clr         i
digloop     call        getdigit
            mov         byte,w
            cje         byte,#$FF,done
            call        @load_frequencies       ; VP routine
            call        @dial_it                ; VP routine
            inc         i
            mov         w,#20
            call        @delay_10n_ms
            jmp         digloop
done
            sleep
```

To dial again, reset the processor. The **load_frequencies**, **dial_it**, and **delay_10n_ms** routines are all part of the VP (and they reside on different pages which explains the at sign prefix). The **getdigit** routine is a simple lookup table that returns the phone number digits.

Mixing VPs

When you need to mix VPs, there are several areas you have to consider:

1. At what frequency must the ISRs run?
2. Port and variable conflicts
3. Conflicting uses of the **!option** register
4. Varying time paths through the ISR

Most of these issues are straightforward. Sometimes you can adjust parameters to resolve conflicts. For example, if you need a UART, you can adjust its timing so that it will work with other VPs that don't use the same frequency. Sometimes it is more difficult and requires significant effort to rewrite the VPs code.

Another issue is varying time paths through the ISR. Some VPs depend on an exact amount of time passing between interrupts. PWM generation, for instance, requires precise timing. If you merge a VP that requires an exact amount of time between interrupts with another VP, you should place the time-sensitive VP's interrupt code before the other VP's code. Reversing this order will upset the sensitive VP if the other VP's ISR does not always require the same time to execute. A few VPs use special techniques to ensure that they always require the same amount of time to execute, but most can take varying times depending on conditions.

Summary

Using VPs you can create powerful programs easily. However, it does take a bit of experience and effort to peel away the interesting parts of the VP examples and apply them to your program. The effort, however, is usually far less than it would take you to duplicate the VPs features in either hardware or software.

You can mix VPs if you are careful. However, blending together VPs can often be taxing as you try to make peace between conflicting requirements for each module.

Exercises

1. Download the DTMF generation VP and remove the portions that are unnecessary for building an auto dial program that automatically dials a phone number when it starts.
2. Move the DTMF output to Port C pin 6.
3. Add your own code to dial a number of your choice each time the processor resets. Put the processor to sleep after dialing. To hear the tones, you can connect a piezo speaker to the port. However, this will probably be too rough and too weak to really dial a phone. If you want to really dial the phone, add an RC filter (see the instructions in the VP documentation; you'll need a 600 ohm resistor and a capacitor around .2uF). You can then use an amplified speaker or signal tracer to increase the volume to where it can really dial the phone.

Answers

Here is the listing that satisfies the three problems in this unit:

```
            device      sx28l,stackx_optionx
            device      oscxt5,turbo

            freq  50_000_000             ; default run speed = 50MHz
            ID    'DIAL'

            reset start                  ; JUMP to start label on reset


;**************
; Equates for common data comm frequencies
;**************
f697_h      equ         $012  ; DTMF Frequency
f697_l      equ         $09d

f770_h      equ         $014  ; DTMF Frequency
f770_l      equ         $090

f852_h      equ         $016  ; DTMF Frequency
f852_l      equ         $0c0

f941_h      equ         $019  ; DTMF Frequency
f941_l      equ         $021

f1209_h     equ         $020  ; DTMF Frequency
f1209_l     equ         $049

f1336_h     equ         $023  ; DTMF Frequency
f1336_l     equ         $0ad

f1477_h     equ         $027  ; DTMF Frequency
f1477_l     equ         $071

f1633_h     equ         $02b  ; DTMF Frequency
f1633_l     equ         $09c


;**************
```

8

```
; Pin Definitions
;**************
;PWM_pin         equ   rb.7        ; DTMF output
PWM_pin          equ   rc.6        ; DTMF output

;*************
;     Global Variables
;*************
                 org   $8          ; Global registers

flags            ds    1
dtmf_gen_en      equ   flags.1     ; Tells if DTMF output is enabled
timer_flag       equ   flags.2     ; Flags a rollover of the timers.
temp             ds    1           ; Temporary storage register
byte             ds    1           ; a byte
i                ds    1           ; loop counter

;*************
;     Bank 0 Variables
;*************
                 org   $10

sin_gen_bank     =     $

freq_acc_high    ds    1           ;
; 16-bit accumulator which decides when to increment the sine wave
freq_acc_low     ds    1
freq_acc_high2   ds    1           ;
; 16-bit accumulator which decides when to increment the sine wave
freq_acc_low2    ds    1
freq_count_high  ds    1           ; freq_count = Frequency * 6.83671552
freq_count_low   ds    1           ; 16-bit counter
;decides which frequency for the sine wave

freq_count_high2 ds    1           ; freq_count = Frequency * 6.83671552
freq_count_low2  ds    1           ; 16-bit counter which
;decides which frequency for the sine wave

curr_sin         ds    1           ; The current value of the sin wave
sinvel           ds    1           ; The velocity of the sin wave

curr_sin2        ds    1     ; The current value of the sin wave
```

```
sinvel2          ds    1           ; The velocity of the sin wave

sin2_temp        ds    1     ; Used to do a temporary shift/add register

PWM_bank         =     $

pwm0_acc         ds    1           ; PWM accumulator
pwm0             ds    1           ; current PWM output

;*************
;     Bank 1 Variables
;*************
                 org     $30       ;bank3 variables
timers           =     $
timer_l          ds    1
timer_h          ds    1

;**************
; Interrupt
;
; With a retiw value of -163 and an oscillator frequency of 50MHz, this
; code runs every 3.26us.
;**************
            org   0
;**************
PWM_OUTPUT
; This outputs the current value of pwm0 to the PWM_pin.  This generates
; an analog voltage at PWM_pin after filtering
;**************
            bank  PWM_bank
            add   pwm0_acc,pwm0     ; add the PWM output to the acc
            snc
            jmp   :carry            ; if there was no carry, then clear
                                    ; the PWM_pin
            clrb  PWM_pin
            jmp   PWM_out
:carry
            setb  PWM_pin           ; otherwise set the PWM_pin
PWM_out
;**************
            jnb   dtmf_gen_en,sine_gen_out
            call  @sine_generator1
```

8

```
sine_gen_out

;**************
do_timers
; The timer will tick at the interrupt rate (3.26us for 50MHz.)  To set up
; the timers, move in FFFFh - (value that corresponds to the time.)
; Example:
; for 1ms = 1ms/3.26us = 306 dec = 132 hex so move in $FFFF - $0132 =
; $FECD
;**************

            bank  timers                 ; Switch to the timer bank
            mov   w,#1
            add   timer_l,w        ; add 1 to timer_l
            jnc   :timer_out       ; if it's not zero, then
            add   timer_h,w        ; don't increment timer_h
            snc
            setb  timer_flag
:timer_out
;**************
:ISR_DONE
; This is the end of the interrupt service routine.
; Now load 163 into w and
; perform a retiw to interrupt 163 cycles from the start of this one.
; (3.26us@50MHz)
;**************
            break
; interrupt 163 cycles after this interrupt
            mov   w,#-163
            retiw                  ; return from the interrupt
;**************

start       bank  sin_gen_bank     ; Program starts here on power up

      ;*****************************************************
      ; Initialize ports and registers
      ;*****************************************************

; use these values for a wave which is 90 degrees out of phase.
            mov   curr_sin,#-4
            mov   sinvel,#-8
```

```
; use these values for a wave which is 90 degrees out of phase.
            mov   curr_sin2,#-4
            mov   sinvel2,#-8
            call  @disable_o

            mov   !option,#%00011111      ; enable wreg and rtcc interrupt
            mov   !rc,#%10111111

            mov   m,#$D                   ; make cmos-level
            mov   !rc,#%10111111
            mov   m,#$F

; load digits
            clr   i
digloop     call  getdigit
            mov   byte,w
            cje   byte,#$FF,done
            call  @load_frequencies       ; load the frequency registers
            call  @dial_it                ; dial the number for 60ms
; and return.
            inc   i
            mov   w,#20
            call  @delay_10n_ms
            jmp   digloop
done
            sleep

; get i'th digit to dial
getdigit    mov   w,i
            jmp   PC+W
            retw  1,8,8,8,5,1,2,1,0,2,4,$FF



org   $200              ; Start this code on page 1
;***********************************************************
;     Miscellaneous subroutines
;***********************************************************
delay_10n_ms
; This subroutine delays 'w'*10 milliseconds.
; This subroutine uses the TEMP register
```

8

```
; INPUT           w     -     # of milliseconds to delay for.
; OUTPUT    Returns after n milliseconds.
;*******************************************************
      mov   temp,w
      bank  timers
:loop clrb  timer_flag  ; This loop delays for 10ms
      mov   timer_h,#$0f4
      mov   timer_l,#$004
      jnb   timer_flag,$
      dec   temp        ; do it w-1 times.
      jnz   :loop
      clrb  timer_flag
      retp

;*******************************************************
; Subroutine - Disable the outputs
; Load DC value into PWM and disable the output switch.
;*******************************************************
disable_o
            bank  PWM_bank    ; input mode.
            mov   pwm0,#128   ; put 2.5V DC on PWM output pin
            retp

org   $400              ; This table is on page 2.
; DTMF tone table
_0_         dw    f941_h,f941_l,f1336_h,f1336_l
_1_         dw    f697_h,f697_l,f1209_h,f1209_l
_2_         dw    f697_h,f697_l,f1336_h,f1336_l
_3_         dw    f697_h,f697_l,f1477_h,f1477_l
_4_         dw    f770_h,f770_l,f1209_h,f1209_l
_5_         dw    f770_h,f770_l,f1336_h,f1336_l
_6_         dw    f770_h,f770_l,f1477_h,f1477_l
_7_         dw    f852_h,f852_l,f1209_h,f1209_l
_8_         dw    f852_h,f852_l,f1336_h,f1336_l
_9_         dw    f852_h,f852_l,f1477_h,f1477_l
_star_      dw    f941_h,f941_l,f1209_h,f1209_l
_pound_     dw    f941_h,f941_l,f1477_h,f1477_l

org   $600              ; These subroutines are on page 3.
;*************************************
; DTMF transmit functions/subroutines
;*************************************
```

```
;************************************
load_frequencies
; This subroutine loads the frequencies using a table lookup approach.
; The index into the table is passed in the byte register.  The DTMF table
; must be in the range of $400 to $500.
;************************************
            cje   byte,#$0FF,:end_load_it
            clc
            rl    byte
            rl    byte               ; multiply byte by 4 to get offset
            add   byte,#_0_          ; add in the offset of the first digit
            mov   temp,#4
            mov   fsr,#freq_count_high

:dtmf_load_ loop  mov   m,#4               ; mov 4 to m (table is in $400)
            mov   w,byte
            IREAD                    ; get the value from the table
            bank  sin_gen_bank             ; and load it into the frequency
            mov   indf,w                   ; register
            inc   byte
            inc   fsr
            decsz temp
            jmp   :dtmf_load_loop    ; when all 4 values have been loaded,
:end_load_it  retp                   ; return
;************************************************
dial_it     ; This subroutine puts out whatever frequencies were loaded
            ; for 1000ms, and then stops outputting the frequencies.
;**************
            cje   byte,#$0FF,end_dial_it
            bank  sin_gen_bank
; use these values to start the wave at close to zero crossing.
            mov   curr_sin,#-4
            mov   sinvel,#-8
; use these values to start the wave at close to zero crossing.
            mov   curr_sin2,#-4
            mov   sinvel2,#-8
            enable_o                 ; enable the output
            mov   w,#3
            call  @delay_10n_ms            ; delay 30ms
            setb  dtmf_gen_en
            mov   w,#10
            call  @delay_10n_ms            ; delay 100ms
```

8

```
            clrb  dtmf_gen_en
            call  @disable_o       ; now disable the outputs
end_dial_it retp
;************************************************
sine_generator1              ;(Part of interrupt service routine)
; This routine generates a synthetic sine wave with values ranging
; from -32 to 32.  Frequency is specified by the counter.  To set the
; frequency, put this value into the 16-bit freq_count register:
; freq_count = FREQUENCY * 6.83671552 (@50MHz)
;**************************************************
            bank  sin_gen_bank
; advance sine at frequency
            add   freq_acc_low,freq_count_low;2
            jnc   :no_carry        ;2,4  ; if lower byte rolls over
            inc   freq_acc_high          ; carry over to upper byte
            jnz   :no_carry              ; if carry causes roll-over
; then add freq counter to accumulator (which should be zero,
; so move will work)
            mov   freq_acc_high,freq_count_high
                                         ; and update sine wave
            jmp   :change_sin
:no_carry
; add the upper bytes of the accumulators
            add   freq_acc_high,freq_count_high
            jnc   :no_change
:change_sin

            mov   w,++sinvel  ;1         ; if the sine wave
            sb    curr_sin.7       ;1    ; is positive, decelerate
            mov   w,--sinvel  ;1         ; it.  otherwise, accelerate it.
            mov   sinvel,w    ;1
            add   curr_sin,w       ;1    ; add the velocity to sin


:no_change

;*************************************************
sine_generator2                    ;(Part of interrupt service routine)
; This routine generates a synthetic sine wave with values ranging
; from -32 to 32.  Frequency is specified by the counter.  To set the
; frequency, put this value into the 16-bit freq_count register:
; freq_count = FREQUENCY * 6.83671552 (@50MHz)
```

```
;***********************************************

;advance sine at frequency
            add   freq_acc_low2,freq_count_low2 ;2
            jnc   :no_carry         ;2,4  ; if lower byte rolls over
            inc   freq_acc_high2    ; carry over to upper byte
            jnz   :no_carry               ; if carry causes roll-over
; then add freq counter to accumulator (which should be zero,
            mov   freq_acc_high2,freq_count_high2
                                          ; so move will work)
                                          ; and update sine wave
            jmp   :change_sin
:no_carry
; add the upper bytes of the accumulators

            add   freq_acc_high2,freq_count_high2
            jnc   :no_change
:change_sin

            mov   w,++sinvel2 ;1          ; if the sine wave
            sb    curr_sin2.7 ;1          ; is positive, decelerate it
            mov   w,--sinvel2 ;1          ; it.  Otherwise, accelerate it.
            mov   sinvel2,w   ;1
            add   curr_sin2,w ;1          ; add the velocity to sin


:no_change
            mov   pwm0,curr_sin2    ; mov sin2 into pwm0
            mov   sin2_temp,w
; mov the high_frequency sin wave's current value
            clc                            ; into a temporary register

; divide temporary register by four by shifting right
            snb   sin2_temp.7
            stc                            ; (for result = (0.25)(sin2))
            rr    sin2_temp
            clc
            snb   sin2_temp.7
            stc
            mov   w,>>sin2_temp
; (1.25)(sin2) = sin2 + (0.25)(sin2)
            add   pwm0,w
```

8

```
; add the value of SIN into the PWM output
            add   pwm0,curr_sin
; for result = pwm0 = 1.25*sin2 + 1*sin
; put pwm0 in the middle of the output range (get rid of negative values)
            add   pwm0,#128
            retp                          ; return with page bits intact
```

The programs and information in this tutorial are presented for instructional value. The programs and information have been carefully tested, but are not guaranteed for any particular purpose. The publisher and the author do not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors or omissions. The publisher and author assume no liability for damages resulting from the use of the information in this tutorial or for any infringement of the intellectual property rights of third parties that would result from the use of this information.

Rev1.2

Appendix A. Instruction Summary

Appendix A from Introduction to Assembly Language with the Scenix SX Microcontroller

© 1999 by Parallax, Inc. All Rights Reserved. By Al Williams, AWC

Processor Control

Instruction	Words	Turbo Cycles	Description
BANK x	1	1	Sets current register bank
MODE x	1	1	Sets I/O mode
NOP	1	1	No operation
PAGE	1	1	Sets current code page
SLEEP	1	1	Puts processor in low power sleep mode

Flow Control

Instruction	Words	Turbo Cycles	Description
CALL	1	3	Call subroutine
CJA	4	4,6	Compare jump above
CJAE	4	4,6	Compare jump above or equal
CJB	4	4,6	Compare jump below
CJBE	4	4,6	Compare jump below or equal
CJE	4	4,6	Compare jump equal
CJNE	4	4,6	Compare jump not equal
CSA	3	3,4	Compare skip above
CSAE	3	3,4	Compare skip above or equal
CSB	3	3,4	Compare skip below
CSBE	3	3,4	Compare skip below or equal
CSE	3	3,4	Compare skip equal
CSNE	3	3,4	Compare skip not equal
DECSZ	1	1,2	Decrement skip zero
DJNZ	2	2,4	Decrement jump not zero
INCSZ	1	1,2	Increment skip zero
IJNZ	2	2,4	Increment jump not zero
JB	2	2,4	Jump if bit set
JC	2	2,4	Jump if carry set
JMP	1	3	Jump
JNB	2	2,4	Jump if bit not set
JNC	2	2,4	Jump if no carry
JNZ	2	2,4	Jump if no zero
JZ	2	2,4	Jump if zero
MOVSZ	1	1,2	Move (with optional inc/dec) skip on zero
RET	1	3	Return from subroutine
RETP	1	3	Return across page
RETW	1	3	Return literal
SKIP	1	2	Skip next instruction
SNB	1	1,2	Skip if bit clear
SNC	1	1,2	Skip if no carry
SNZ	1	1,2	Skip if not zero

Math and Logic

Instruction	Words	Turbo Cycles	Description
ADD	1	1	Add (register + W or W + register)
ADD	2	2	Add (register + register or literal)
ADDB	2	2	Add bit

AND	1	1	And (register and W, W and register, W and literal)
AND	2	2	And (register and literal or register and register)
DEC	1	1	Decrement
INC	1	1	Increment
NOT	1	1	Invert
OR	1	1	Or (register and W or W and register or W and literal)
RL	1	1	Rotate left
RR	1	1	Rotate right
SUB	1	1	Subtract W from register
SUB	2	2	Subtract register from register or literal from register
XOR	1	1	Exclusive Or register and W or W and register
XOR	2	2	Exclusive Or register and register or register and literal

Interrupt Handling

Instruction	Words	Turbo Cycles	Description
RETI	1	3	Return from interrupt
RETIW	1	3	Return from interrupt and add W to rtcc

Bit Manipulation

Instruction	Words	Turbo Cycles	Description
CLC	1	1	Clear carry
CLRB	1	1	Clear bit
CLZ	1	1	Clear zero
MOVB	4	4	Move bit
SETB	1	1	Set bit
STC	1	1	Set carry
STZ	1	1	Set zero

Move/Clear/Test

Instruction	Words	Turbo Cycles	Description
CLR	1	1	Clear register, W, or WDT
MOV	1	1	Move W to register, register to W, literal to W
MOV	2	2	Move register to register or literal to register
TEST	1	1	Test W or register, set flags

Miscellaneous

Instruction	Words	Turbo Cycles	Description
IREAD	1	4	Reads program memory
LCALL	1-4	3-6	Obsolete
LJMP	1-4	3-6	Obsolete
LSET	0-3	0-3	Obsolete

The programs and information in this tutorial are presented for instructional value. The programs and information have been carefully tested, but are not guaranteed for any particular purpose. The publisher and the author do not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors or omissions. The publisher and author assume no liability for damages resulting from the use of the information in this tutorial or for any infringement of the intellectual property rights of third parties that would result from the use of this information.

Rev1.2

Appendix B. Hardware

Appendix B from Introduction to Assembly Language with the Scenix SX Microcontroller

© 1999 by Parallax, Inc. All Rights Reserved. By Al Williams, AWC

The projects in this tutorial are simple to build using common components. For the maximum flexibility, you'll want to use a solderless breadboard. If you use the Parallax SX-Tech board you can simply connect the circuits to the integrated breadboard.

You can also use your own breadboard if you like. The SX chip simply requires a regulated 5 volt supply (a bench supply will work fine) and a connection to the SX-Key programmer. If you are using an SX-Blitz, or you want to operate the circuit without the SX-Key, you'll also need a 50MHz ceramic resonator (Murata CST50.00MXW040 or equivalent).

To successfully complete the tutorial exercises, you only need a few common parts:

- LEDs (or 5V LEDs with integrated resistors)
- 470 ohm resistors (if not using 5V LEDs)
- Push button switches
- Non-critical pull up resistors (10K to 22K, 1/4W or 1/8W)
- A piezo electric speaker

Common Circuit

All the circuits require the SX to be connected to the programmer and the chip's support circuitry. Again, if you are using an SX-Tech board this is already done. If you are using the SX-Key, you only need to connect the chip to 5V, ground, and the SX-Key. You can use an existing 5V power supply if you have one (make sure it is regulated). If you want to build a simple 5V supply, look at figure B.1. This supply will handle about 100mA as shown, or can handle over 1A if you use a 7805 with a heat sink in place of the 78L05 specified. You can use an ordinary wall transformer to supply the unregulated DC input.

To ensure proper operation, you should also connect the MCLR pin to 5V either directly or through a pull up resistor. If you use a pull up resistor you'll be able to short the MCLR pin to ground to reset the processor. For the ultimate convenience you could use a push button switch to make the ground connection.

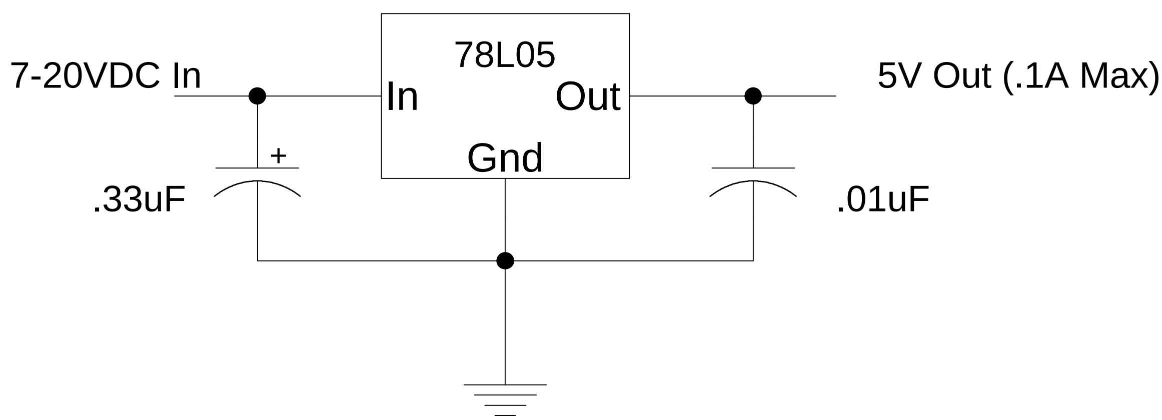

Figure B.1 A Simple 5V Supply

To connect the programmer, you can use pins with .1 inch spacing. You usually buy these in strips that you can snap to the correct length with a pair of pliers. Insert one end into your breadboard and the SX-Key (or SX-Blitz) will plug into the other side. If one side of the pins is too short, you can usually slide the plastic insulator with a pair of pliers so that the pins on each side are of equal length. Table B.1 shows the pin connections necessary.

	5V	Ground	OSC1	OSC2	MCLR
SX18	14	5	16	15	4
SX28	15,16	5,6	18	17	4

Table B.1 SX Pin Connections

I/O Circuits

Most of the projects in the tutorial require some input or output. The I/O usually takes the form of an LED, a push button, both an LED and a push button, or a piezo speaker. Figure B.2 shows the common LED hookup. If you are using 5V LEDs, you don't need the resistor as it is built into the LED. Notice that the LED is polarized; refer to the LEDs specifications to identify which lead is which. With the LED wired as shown, you must bring the SX pin low to light the LED.

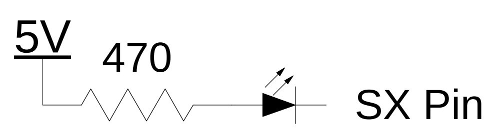

Figure B.2 An LED Circuit

In unit 5, some exercises use a push button and a piezo speaker for I/O (see Figure B.3). The 10K resistor's value is not overly critical. Anything from 10K to 22K (or even more) should work fine. If a project calls for more switches, you can duplicate the switch portion of the circuit for other pins. Just use a pull up resistor on the pin and connect the switch to ground.

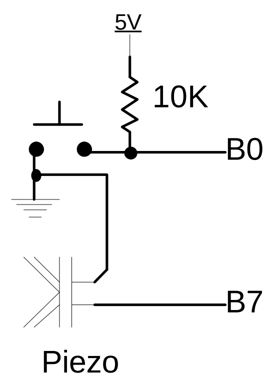

Figure B.3 A Speaker and Switch Circuit

Don't connect an ordinary speaker directly to the SX pin as the load presented by such a speaker may damage the SX chip. Most ICs, including the SX, can directly drive a piezo speaker.

About the SX Demo Board

If you have one of the older SX Demo Boards, all the circuitry you need for these exercises is already present on the board. In Unit 7, some of the programs use a combination switch and LED, as you will find on the SX Demo Board (see Figure B.4). However, this circuit works best when the internal pull up resistors are turned on for the SX pins that connect to it.

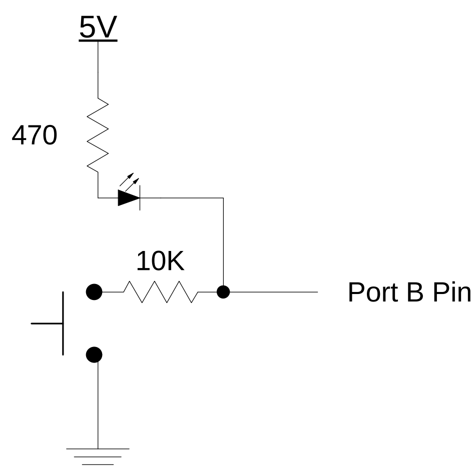

Figure B.4 – Switch/LED Combination

The Final Project

The final project in this tutorial is a TouchTone phone dialer. For demonstration purposes, you can hear the tones in a piezo speaker (although they may be quite low – you may have to put your ear right up to the speaker). If you want to really dial a phone, you'll need two things: a filter and an amplifier.

The Scenix notes on the DTMF generation VP specifies the component values for the low pass filter. This filter prevents high-frequency noise (an unavoidable byproduct of using PWM to generate tones) from entering the phone lines. Connect a 620 ohm resistor to the SX output pin and a .22uF capacitor from the other side of the resistor to ground (the Scenix data calls for 600 ohms and .2uF capacitors, but these values are close enough and easy to obtain). This will make the tones even weaker than before, however. Some sort of amplification is necessary if you plan to feed the tones into the phone. You can use any sort of amplified speaker, signal tracer, or build a small amplifier from an LM386 chip (see Figure B.5) and drive an ordinary 8 ohm speaker.

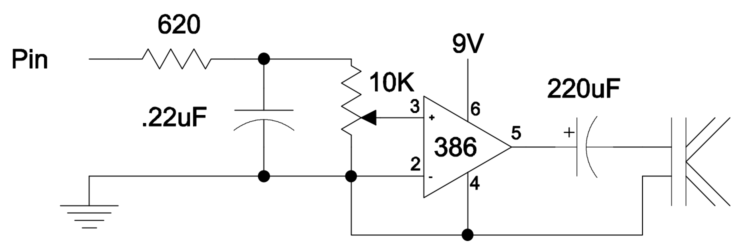

Figure B.5 – A Simple Amplifier

The programs and information in this tutorial are presented for instructional value. The programs and information have been carefully tested, but are not guaranteed for any particular purpose. The publisher and the author do not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors or omissions. The publisher and author assume no liability for damages resulting from the use of the information in this tutorial or for any infringement of the intellectual property rights of third parties that would result from the use of this information.

Rev1.2