Introducción a la Programación en Lenguaje Assembler con el Microcontrolador Scenix SX

Guía Educativa para el Programa Universitario SX

Versión en Castellano 1.0

Garantía

Parallax garantiza sus productos contra defectos en sus materiales o debidos a la fabricación por un período de 90 días. Si usted descubre un defecto, Parallax según corresponda, reparará, reemplazará o regresará el valor de la compra. Simplemente pida un número de autorización de regreso de mercadería (Return Merchandise Authorization, "RMA"), escriba el número en el exterior de la caja y envíela a Parallax. Por favor incluya su nombre, número telefónico, dirección, y una descripción del problema. Nosotros le regresaremos su producto o el reemplazo, usando el mismo método de correo que usted usó para enviar el producto a Parallax.

Garantía de 14 días de regreso de dinero

Si dentro de los 14 días en que usted recibió su producto, encuentra que no es conveniente para sus necesidades, puede regresarlo, recibiendo un reembolso. Parallax regresará el precio de compra del producto, excluyendo los costos de manipuleo y correo. Esto no se aplica si el producto a sido alterado o dañado.

Derechos de Copia y Marcas Registradas

Esta documentación tiene derechos de copia Copyright 1999 por Parallax, Inc. BASIC Stamp (Estampilla BASIC) es una marca registrada de Parallax, Inc. Si usted decide usar el nombre "BASIC Stamp" en su página web o en material impreso, debe agregar la aclaración: "BASIC Stamp es una marca registrada de Parallax, Inc." Otros nombres de productos son marcas registradas de sus respectivos dueños.

Desvinculación de Responsabilidad

Parallax, Inc. no es responsable de daños por consecuencias, incidentes o daños especiales que resulten de cualquier violación de la garantía, bajo cualquier teoría legal, incluyendo pérdida de beneficio, tiempo, daño o reemplazo de equipo o propiedad y cualquier costo, recuperando, reprogramando o reproduciendo cualquier dato guardado o usado dentro de los productos Parallax. Parallax tampoco es responsable de cualquier daño personal, incluyendo vida o muerte, resultado del uso de cualquiera de nuestros productos. Usted tiene absoluta responsabilidad por la aplicación que desarrolle con el BASIC Stamp.

Acceso en Internet

Mantenemos sistemas de Internet para su uso. Estos pueden ser usados para obtener software, comunicarse con miembros de Parallax, y comunicarse con otros clientes. Las rutas de acceso a la información se muestran a continuación:

E-mail: stampsinclass@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

Contenido

Unidad I. Comenzando **7**
Sobre este curso 7
Comience por el Principio 8
Problema 1 9
Problema 2 10
Cuide su Lenguaje 10
El Ambiente de Trabajo 11
¿Eso es todo? 11
El Ciclo de Desarrollo 12
Sistemas Numéricos 13
Otros lugares, otras bases 14
Aclarando la base 15
Tamaño 15
La Conexión del Hardware 15
Resumen 16
Ejercicios 16
Respuestas 17
Unidad II. Su Primer Programa **19**
Primer Paso 19
Ejecutar el Programa 20
¿Y Ahora Qué? 21
Dentro del Programa 21
Registros 22
Depuración Elemental 23
Deteniendo el Depurador 26
Resumen 27
Ejercicios 27
Respuestas 28
Unidad III. Control de Flujo Simple **29**
Contador de Programa 29
¿Más Interesante? 30
¿Qué está mal? 31
Otras Formas de JMP 32
Rótulos Locales 33
Otra Forma de INC 33
Deteniendo el Controlador 34
Sobre el Watchdog 34
Resumen 36
Ejercicios 37
Respuestas 38
Unidad IV. Variables y Matemática **39**
Un Ejemplo 40
Asignación 42

Mejorando la Matemática ... 43
Números en Complemento a Dos ... 44
Más Trucos de Acarreo ... 45
¡Pruébelo! ... 45
Algunas Funciones Más ... 46
Pausas Programadas ... 48
Funciones Lógicas ... 49
Resumen ... 52
Ejercicios ... 52
Respuestas ... 53
Unidad V. Control de Flujo Avanzado ... 55
Comparando ... 56
Usando Call y Return ... 57
Tablas ... 61
Funciones Matemáticas ... 63
División ... 64
Resumen ... 66
Ejercicios ... 66
Respuestas ... 68
Unidad VI. Programación de Bajo Nivel ... 73
Control de Puertos ... 73
Capacidades Analógicas ... 75
Banco de Registros ... 75
Páginas de Programa ... 78
Leyendo un Programa Almacenado ... 79
Resumen ... 80
Ejercicios ... 80
Respuestas ... 81
Unidad VII. Interrupciones ... 83
El Contador de Reloj de Tiempo Real ... 83
Pausas RTCC ... 85
Interrupciones RTCC ... 85
Interrupciones Periódicas ... 87
Un Ejemplo de Reloj ... 88
Interrupciones Externas mediante RTCC ... 90
Detección de Flancos de Entrada del Puerto B ... 90
Interrupciones del Puerto B ... 94
Resumen ... 94
Ejercicios ... 94
Respuestas ... 95
Unidad VIII. Periféricos Virtuales ... 101
Usando un Periférico Virtual ... 102
Mezclando Periféricos Virtuales ... 103
Resumen ... 104

Ejercicios 104
Respuestas 105
Apéndice A. Resumen de Instrucciones 115
Apéndice B. Hardware...123

Unidad I. Comenzando

Unidad I de Introducción al Lenguaje Assembler con el Microcontrolador Scenix SX

© 1999 Parallax, Inc. Reservados todos los derechos. Autor Al Williams, AWC. Traducción Arístides Álvarez.

Allá por 1943, el presidente de IBM dijo que llegaría el día en el que habría un mercado mundial para cinco computadoras. Hoy, las computadoras están por todos lados. Por supuesto que hay PC en muchas casas, pero la verdadera invasión de las computadoras no está en las PC hogareñas. La gente compra computadoras en cada dispositivo electrónico que adquiere. En la actualidad su televisor, su teléfono, su horno microondas, y su auto tienen computadoras (algunos tienen varias).

Estas computadoras, obviamente, pueden no ser tan poderosas como su PC de escritorio, pero están diseñadas para controlar el "mundo real". Un hecho indispensable en el diseño de equipamiento electrónico actual (por diversión o por negocios) es comprender cómo trabajan estos dispositivos y cómo puede usarlos en sus propias creaciones.
¿Por qué usar estos microcontroladores? A menudo un microcontrolador puede reemplazar una gran cantidad de componentes. Por ejemplo, considere un contestador automático. ¿Es indispensable el uso de un microcontrolador para realizar esta tarea? No. De hecho, muchos de los contestadores antiguos no usaban microcontroladores. Tenían un circuito para detectar la llamada. Este circuito activaba un temporizador (o en una máquina realmente vieja un disco conectado a un motor). El temporizador conectaba un relé que "descolgaba" el teléfono. Luego otro temporizador reproducía un mensaje indicador grabado en una cinta. Una vez finalizado este mensaje (basándose en el tiempo transcurrido, o sensando un silencio largo en el fin del mensaje), otro temporizador grababa la llamada.

En lugar de tres temporizadores, los contestadores actuales usan un microcontrolador. Con pocos componentes externos, el microcontrolador puede operar todo el sistema con facilidad. Pero hay mucho más. Un microcontrolador también puede sensar si alguien está hablando al otro lado de la línea. Puede aceptar comandos Touch-Tone (por tonos DTMF) para permitir control remoto. Incluso puede almacenar y reproducir la voz digitalmente, en lugar de con cintas. Intente hacer un control remoto sofisticado sin un microcontrolador.

Así que nuestro contestador microcontrolado es mucho más poderoso que sus ancestros. Además cuesta menos. Los microcontroladores son bastante baratos (incluso sin tener en cuenta la cantidad de componentes que pueden reemplazar). Con menos componentes se logran dispositivos más pequeños, baratos, y menos propensos a fallas.

Sobre este curso

Este curso está completamente dedicado a la incorporación de estas pequeñas y poderosas computadoras (microcontroladores) en sus propios diseños. En particular usaremos el microcontrolador Scenix SX junto con el sistema de desarrollo SX-Key de Parallax. El SX es un microcontrolador barato pero muy poderoso. El SX-Key le permite programar el SX y también depurar sus programas en tiempo real. En el pasado, hardware como el SX-Key era muy caro (miles de dólares) y sólo estaba disponible en laboratorios bien provistos. Sin embargo, el SX-Key es muy accesible (solamente unos cientos de dólares, dependiendo de las opciones).

Para obtener el máximo provecho de este curso, debería estar familiarizado con el diseño de hardware básico. Debería comprender cómo funcionan los LEDs, por ejemplo, y comprender leyes básicas de la electrónica (como la ley de Ohm). Este curso se enfocará en el diseño de programas a ser ejecutados en el microcontrolador, controlando así circuitos electrónicos. Aunque normalmente se piensa en los programas como software, cuando un programa está dentro de un microcontrolador, a menudo se lo llama firmware (una cruza entre hardware y software).

Los laboratorios de este curso son más fáciles de realizar con la SX-Tech Board disponible en Parallax. Sin embargo, usted puede montar su propia versión de estos circuitos en cualquier protoboard o plaqueta experimental (Ver Apéndice B).

El chip SX es muy poderoso, pero también es muy útil como herramienta de aprendizaje. A diferencia de otros microcontroladores, el SX usa memoria que se borra eléctricamente para almacenar programas. Esto significa que puede escribir un programa, probarlo, y luego reprogramar el chip inmediatamente para ejecutar un programa diferente (o una versión corregida del mismo programa). Esto acoplado con las poderosas herramientas del SX-Key provee un ambiente ideal para el aprendizaje y la experimentación.

Comience por el Principio

Si no está familiarizado con el funcionamiento interno de una computadora, puede parecerle magia negra. Puede parecer que el pequeño chip puede hacer prácticamente cualquier cosa, sin importar la complejidad. Sin embargo, detrás de esta complejidad hay una sorpresa. El microcontrolador opera de una manera muy simple. Esta simplicidad significa que usted (el programador) debe realizar un gran esfuerzo para crear estos comportamientos complejos. La programación requiere pensamiento lógico y atención a los detalles.

Todos los programas funcionan usando una secuencia almacenada de instrucciones. Estas instrucciones le dicen a la computadora qué hacer. Cuando la computadora se inicia, lee estas instrucciones en secuencia. Algunas instrucciones leen entradas. Otras controlan salidas. Otras instrucciones realizan cierto tipo de procesamiento.

El Scenix SX posee una arquitectura Harvard. Esto significa que hay un área que recuerda las instrucciones y otra que almacena los datos (incluyendo entradas y salidas). Esta es una arquitectura común para los microcontroladores (aunque otras computadoras utilizan la arquitectura Von Neumann donde se mezclan los datos con las instrucciones).
Imagine que comienza en un trabajo nuevo en una fábrica de radios. El capataz de la planta le da las siguientes instrucciones:

1. Ponga una caja vacía debajo de la cinta transportadora
2. Presione el interruptor rojo para encender la cinta transportadora
3. Observe como caen las radios dentro de la caja. Por cada radio que cae presione su contador de mano.
4. Cuando el contador llegue a 10, presione nuevamente el interruptor rojo para detener la cinta transportadora
5. Reemplace la caja por una vacía
6. Ponga a cero el contador de mano
7. Vuelva al paso 2

Esto es exactamente igual a un programa de computadora. Es una secuencia de pasos. Tiene entradas (usted decide que una radio ha salido por la cinta transportadora). Tiene salidas (usted presiona el botón rojo, por ejemplo). También tiene procesamiento en la toma de decisiones. De hecho, este es el tipo de trabajo para el que las computadoras fueron hechas.

Problema 1

Hay un pequeño problema. Fuera de la computadora de Star Trek (Viaje a las Estrellas) las computadoras no comprenden instrucciones comunes como éstas. ¿Cómo instruir a la computadora para realizar estos pasos? Cada computadora, desde el microcontrolador más pequeño hasta la supercomputadora más grande, almacenan sus instrucciones en forma de números. Peor aún, almacenan estos números usando aritmética base 2 (binaria, tema que trataremos en esta unidad). Esto significa que el programa de computadora se ve como una serie de 1s y 0s. Este es llamado lenguaje de máquina, y es la base de cada programa de computadora.

Por supuesto, los números base 2 no son fáciles de entender para los humanos, así que se escogió escribir los números con un sistema más manejable. Sin embargo, sigue siendo difícil comprender un programa escrito solamente con números. Por esta razón, los ingenieros usan normalmente un método más conveniente para expresar los programas.
La forma más común de programar microcontroladores es usando el *lenguaje assembler*. Este es un método directo que utiliza abreviaturas en lugar de unos y ceros. Podría usar instrucciones como **ADD** (sumar) o **JMP** (jump o saltar). Antiguamente debía convertir manualmente este código en unos y ceros, pero en la actualidad un programa especial llamado *assembler* (ensamblador) lo hace por usted. Por supuesto, el microcontrolador no puede ejecutar este programa, pero su PC si. A menudo se denomina *cross assembling* al hecho de usar una computadora para ensamblar (convertir las abreviaturas o "mnemónicos" en unos y ceros) el código para otra computadora.

A mucha gente le resulta intimidante el hecho de usar estas instrucciones de bajo nivel. Aunque los mnemónicos son fáciles de leer, siguen representando al lenguaje de máquina, que es muy simple. Por ejemplo, un típico microcontrolador no puede multiplicar o dividir números directamente. Estas operaciones deben ser implementadas mediante sumas y restas. Por esta razón, algunos programadores emplean lenguajes de alto nivel como Basic o C, que pueden resultarles conocidos de otros sistemas de computación.

¿Si se puede usar Basic o C para programar microcontroladores, por qué usar assembler o código de máquina? La respuesta es eficiencia. Los microcontroladores tienen generalmente cantidades limitadas de memoria. Además, a menudo se necesita que funcionen lo más rápido posible. Un programa que usa un lenguaje de alto nivel, necesitará más memoria que un programa correctamente escrito en assembler y se ejecutará más lentamente.
Si usa Basic o C, realizará la mayor parte de su trabajo en su PC. Usted escribe su programa en C en la PC y ésta convierte su programa en código de máquina. Parallax fabrica un producto muy exitoso llamado Basic Stamp que usa la PC para convertir código Basic en un cuasi-código de máquina. El Basic Stamp luego ejecuta un programa que interpreta este código, cumpliendo con los pasos programados.

Tip: Los distintos microcontroladores tienen distintos lenguajes de máquina. Sin embargo, una vez que aprende el lenguaje de uno, los otros son relativamente fáciles de dominar.

Problema 2

El siguiente problema es qué hacer una vez que tenemos los ceros y unos. De algún modo deben ser grabados o cargados en el microcontrolador. Los microprocesadores antiguos usaban un chip de memoria externo, pero los microcontroladores modernos tienen una memoria interna que se programa mediante un dispositivo especial llamado *programador*. La memoria de algunos microcontroladores se borra con luz ultravioleta, pero el SX se reprograma instantáneamente, sin necesidad de esperar que alguna luz especial borre su memoria.

En un microcontrolador con arquitectura Harvard, el código del programa no puede ser cambiado o modificado mientras se está ejecutando. Muchos microcontroladores no pueden leer datos de su programa mientras lo están ejecutando. Sin embargo, el SX tiene una característica especial que le permite leer valores de la memoria del programa mientras está funcionando. Esto puede ser útil para almacenar constantes, por ejemplo.

Cuide su Lenguaje

En este curso, usará lenguaje assembler para programar el SX. Sin embargo, si está familiarizado con Basic o C encontrará ejemplos paralelos en estos códigos, para ayudarle a comprender el código assembler.

El Basic Stamp de Parallax usa una versión especial de Basic conocida como Pbasic. El código Basic imitará el lenguaje del Stamp para que pueda aplicar los mismos conceptos con él. Hay algunas variantes del Basic Stamp y una de ellas tiene dentro un microcontrolador SX. Sin embargo, debe programar el Stamp usando Pbasic, no puede usar lenguaje de máquina. Por otro lado, se debe preguntar por qué usar lenguaje de máquina si podría usar Basic. La verdad es que el Basic es genial, pero algunos trabajos necesitan la velocidad y las capacidades que solamente se pueden obtener con código de máquina.

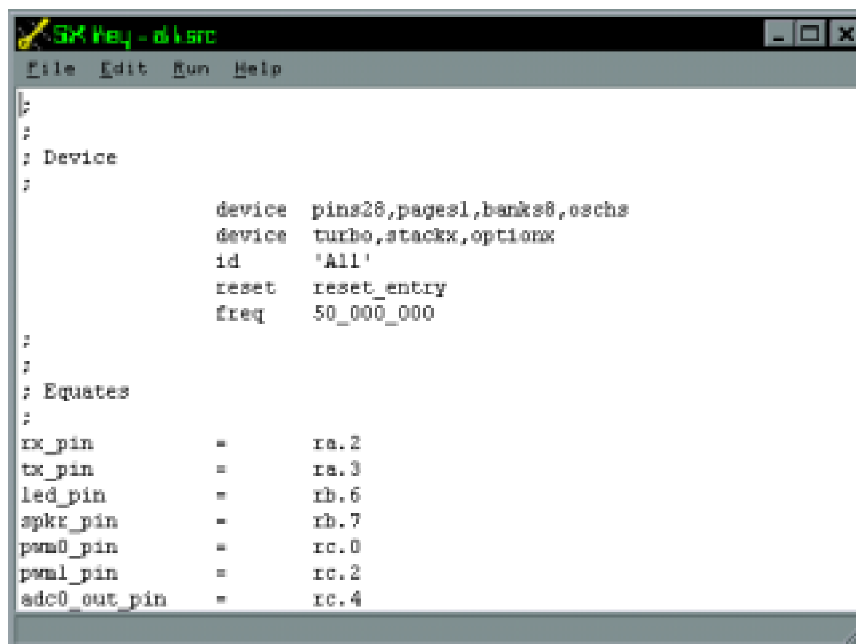

Figura I.1 – El Editor SX-Key

El Ambiente de Trabajo

La Figura I.1 muestra la pantalla principal del SX-Key. Parece un editor de texto común, y en este punto lo es. Puede ingresar código en lenguaje assembler en esta ventana. Cuando quiera probar o ejecutar su programa puede emplear el menú Run para controlar la sintaxis o programar su chip SX. Para buscar errores simples en su código, use el menú Run|Assemble. También puede usar Run|Run para ejecutar el código (asumiendo que tenga el chip conectado al hardware SX-Key).

Tip: El comando assemble controla solamente errores de sintaxis simples. Usted debe encontrar los errores lógicos (con la ayuda del depurador o debugger).

¿Eso es todo?

El verdadero poder del SX-Key no es el ingreso del código. La parte que destaca es cuando su código no funciona como era de esperarse. En ese caso puede usar el comando Run|Debug.

El depurador o debugger (ver Figura I.2) le permite observar el SX mientras ejecuta el programa un paso a la vez y examinar su operación interna. Si está usando el SX-Blitz, solamente puede programar el SX. El Blitz no tiene la característica de depuración.

El Ciclo de Desarrollo

Como puede imaginar, una herramienta tan poderosa simplifica mucho la programación. Sin embargo usted aún necesita un plan. Hay un antiguo dicho: "La gente no planea fallar, falla en planear." Esto es especialmente cierto en el caso de la programación.

Anteriormente hemos dicho que los programas leen entradas, las procesan, y producen una salida. Este es un buen lugar para comenzar a diseñar su software. Proyectos más complejos requerirán técnicas de diseño más rigurosas, pero muchas veces esta aproximación simple bastará. Sin embargo, casi todos los programas (especialmente los de los microcontroladores) seguirán este modelo. Identificar sus entradas, salidas, y el proceso necesario, es un primer paso sólido hacia la realización de su diseño.

El siguiente paso depende de su entorno, experiencia, y gustos personales. Podría comenzar haciendo una lista de instrucciones similares a las líneas de assembler, mencionadas anteriormente. Algunas personas prefieren realizar el diagrama de flujo de sus programas.

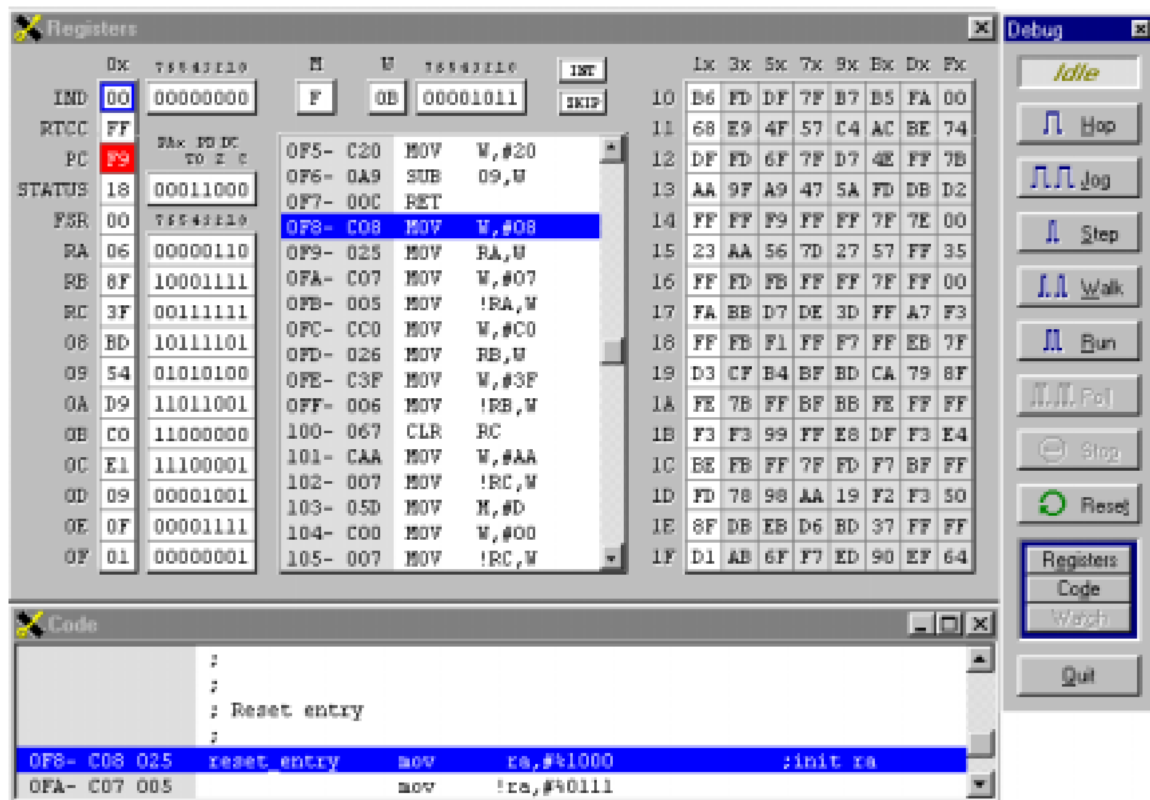

Figura I.2 – El Depurador del SX-Key en Acción

Una vez que tiene una idea de cómo se verá su programa, puede dar el primer paso, ingresando el programa en el editor del SX-Key usando las instrucciones en lenguaje assembler que aprenderá en las unidades siguientes. Puede que el programa funcione en el primer intento, pero es muy improbable. Cuando las cosas no suceden como estaba planeado, ingrese en el depurador para comprender mejor la operación de su programa.

Aunque su programa funcione puede usar el depurador para estudiar su operación. Algunas veces descubrirá mejoras, que no se le ocurrieron mientras pensaba en el programa en forma abstracta.

Sistemas Numéricos

Cuando una persona cuenta, usa el sistema de numeración decimal o base 10. Sin embargo, las computadoras usan números binarios o base 2. Los programadores deben pasar de un sistema a otro, así como también usar otros sistemas.

Cuando usted dice 138 (en decimal) realmente quiere decir:

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

Los dígitos decimales van del 0 al 9.

Los números binarios son parecidos, pero usan solamente dos dígitos: 0 y 1. El número binario 1001 es:

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

Puede ver que simple es pasar de binario a decimal. Sólo debe recordar que cada dígito vale el doble del que está a la derecha.

Ejemplo:

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

En el sentido inverso es un poco más difícil. El truco es determinar cuál es el mayor dígito binario (conocido como *bit*) necesario para representar el número. Considere el número decimal 122. El bit del extremo derecho del número binario siempre vale 1. El siguiente bit vale 2 luego 4, 8, 16, 32, 64, 128, y así.

Debido a que 128 es mayor que 122, ese bit no puede estar en el equivalente binario. Por convención, el bit del extremo derecho se llama bit 0 y los otros bits se numeran secuencialmente de derecha a izquierda. Así el bit con un valor de 128 es el bit 7.

Sin embargo, el bit 6, con un valor de 64, tendrá un 1 en la respuesta debido a que 64 es menor que 122. Dado que 122-64=58 aún debe tener en cuenta esta cantidad. El siguiente valor de bit es 32 que es menor que 58, el bit 5 también tendrá un 1. El resto es 58-32=26.

El bit 4 tiene un peso de (vale) 16 así que también tendrá un 1, dejando un resto de 10. El bit 3 (8) también tendrá un 1, dejando un resto de 2. Ahora considere el bit 2. Tiene un valor de 4 que es mayor que el valor del resto, así que tendrá un 0. El siguiente bit vale 2 así que tendrá un 1 y dejará un resto de 0. Por lo tanto, todos los bits de la derecha (en este caso solamente el bit 0) valdrán cero.

Así que la respuesta es 122 = 1111010. Puede controlarlo invirtiendo la conversión. En otras palabras:

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

Como es evidente, puede agregar todos los ceros que quiera a la izquierda de un número binario (o cualquier número para el caso). Así 1111010, 01111010 y 0000000001111010 son el mismo número.

Otros lugares, otras bases

Debido a que la mayoría de las personas usan el sistema decimal usted ya debe conocerlo. Pero algunas veces es más conveniente usar otras notaciones más simples de convertir a binario. La alternativa más común es la *hexadecimal* o base 16.

La base hexadecimal (conocida como *hex*) usa 16 dígitos, del 0 al 9 y de A a F. Puede observar los valores en la tabla I.1. Observe que para convertir de binario a hex puede simplemente usar la tabla. Así F3 hex es 11110011 binario.

En hexadecimal cada dígito vale 16 veces el anterior. Así F3 hex es:

$15\times16+3\times1=243$

Y 64 hex es:

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

Hex	Decimal	Binario
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

Tabla I.1 – Dígitos Hexadecimales

Tip: Muchas calculadoras, incluido el programa CALC en Windows, pueden convertir entre bases.

Aclarando la base

Con todas estas formas diferentes de escribir números, es fácil confundirse. Incluso el Assembler del SX-Key no puede adivinar mágicamente qué sistema numérico está usando. Este es el motivo por el cual es importante especificar qué clase de número está escribiendo.

Para indicar el sistema numérico en uso, se usan prefijos especiales. Un número que comienza con $, por ejemplo, es un número hexadecimal. Los números binarios comienzan con el caracter %. Debido a que los números decimales son los más comunes, no tienen ningún prefijo.

Tip: No todos los ensambladores usan esta convención. Por ejemplo, algunos usan sufijos para indicar el tipo de número. Otros usan prefijos diferentes. Sin embargo, el assembler del SX-Key que usará en este curso emplea los prefijos que se indicaron.

Tamaño

Otro tema que concierne a los números es cuántos bits ocupan. El SX usa un tamaño de palabra de 8-bits para los datos. A menudo se lo llama un *byte*. El problema con los bytes es que solamente pueden almacenar un valor de 0 a 255. ¿Qué pasa si necesita números más grandes? ¿O números negativos? En esos casos necesitará emplear técnicas especiales que verá en la Unidad VI.
Recuerde, por convención, los bits se numeran empezando por el del extremo derecho. De esa forma el bit del extremo derecho siempre es el bit 0. El bit del extremo izquierdo de un byte es el bit 7. Esto es un poco confuso porque el bit 7 en realidad es el octavo bit (debido a que comienza a contar desde 0 en lugar de 1).
Curiosamente, aunque el SX usa un tamaño de palabra de 8-bits para los datos, sus instrucciones son de 12 bits. Debido a que la arquitectura Harvard separa el código de los datos , esto no es un problema, como parecería en un principio.

La Conexión del Hardware

El SX-Key sólo es una herramienta para lograr un objetivo: ¡programar el chip SX!

El SX es un controlador especialmente rápido. Puede funcionar a velocidades de hasta 100MHz y ejecutar la mayoría de sus instrucciones en un solo ciclo de reloj (10nS por instrucción). En un proyecto real, debe agregar un cristal o un resonador cerámico para velocidades mayores de 4MHz. Sin embargo, cuando trabaja con el SX-Key, éste provee el reloj (puede cambiar la velocidad del reloj usando el menú Run|Clock).

El SX viene en un encapsulado de 18 pines (patitas) o en otra versión de 28 pines. El dispositivo de 18-pines tiene 12 pines de E/S (entrada/salida) y el de 28 pines cuenta con 20 pines de E/S. Ambos dispositivos tienen 2K para almacenar programas y aproximadamente 136 bytes para datos (aunque dispositivos futuros podrán variar estas cantidades de memoria). También hay un modelo de montaje superficial, es un dispositivo de 20 pines que es prácticamente igual al SX de 18 pines. Cuando escribe un 1 en un pin de salida, genera (aproximadamente) 5V. Si escribe un 0 en el pin, genera 0V. Como entrada, el pin reconoce las tensiones superiores al umbral (típicamente 1.4V) como un 1 y las inferiores como un 0. Puede configurar cualquier pin como entrada o salida, e incluso modificarlo durante la ejecución del programa.

Obviamente, la elección del dispositivo dependerá a menudo de la cantidad de pines de E/S que necesite. Si usted quiere usar, por ejemplo, 4 pines para manejar un display LCD, y 8 pines para conectar un teclado, no le quedaría ningún pin para otra tarea si escoge el SX de 18 pines. Sin embargo, en este curso puede estar obligado a usar el que le permita la plaqueta de experimentación, debido a que puede tener el zócalo para un dispositivo en particular.

Puede encontrar detalles del hardware del SX en la hoja de datos oficial. Sin embargo, también leerá más del hardware del SX en el resto de este curso.

Resumen

Un viejo refrán dice: "El árbol gigantesco comienza siendo una pequeña semilla." De manera similar, las simples funciones de un microcontrolador pueden construir sistemas complejos si se las sabe utilizar correctamente.

Para entender las computadoras de bajo nivel, como los microcontroladores, se debe hablar en su idioma, o por lo menos en el lenguaje assembler y códigos hex que la mayoría de las personas usan para representar el lenguaje de máquina secreto.

Esta unidad (por necesidad) cubre estos temas fundamentales. A esta altura debe estar desesperado por comenzar a trabajar con el hardware. Tendrá esta oportunidad en la próxima unidad.

Ejercicios

1. Convierta los siguientes números a decimal:
   (a) $27
   (b) %101110
   (c) $F1
   (d) $AA

2. Convierta los siguientes números a hexadecimal:
   (a) 100
   (b) 200
   (c) 17
   (d) %10110110
   (e) %1000001

3. Conteste Verdadero o Falso a las siguientes afirmaciones:
   (a) Los programas consisten en una serie de pasos o instrucciones.
   (b) Todas las computadoras usan arquitectura Harvard.
   (c) Una computadora con arquitectura Harvard usa memorias separadas para datos y programas.

Respuestas

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

Los programas y la información de este texto son para propósitos educativos, habiendo sido cuidadosamente probados, pero no están garantizados para ningún propósito en particular. El editor, el autor y el traductor no ofrecen ninguna garantía sobre la precisión, adecuación, o completitud de la información presentada, y no asumen ninguna responsabilidad por errores u omisiones en el texto. Tampoco asumen la responsabilidad por daños emergentes por el uso de la información de este texto, o por cualquier violación de los derechos intelectuales o de autor de terceros, que resulten del uso de esta información.

Versión en Castellano 1.0

Unidad II. Su Primer Programa

Unidad II de Introducción al Lenguaje Assembler con el Microcontrolador Scenix SX

© 1999 Parallax, Inc. Reservados todos los derechos. Autor Al Williams, AWC. Traducción Arístides Álvarez.

En este punto, usted probablemente esté listo para comenzar un proyecto. Bien, esto es exactamente lo que hará en esta unidad. Debería tener una PC con el software del SX-Key conectada a la Placa SX-Tech. Si no tiene una Placa SX-Tech puede utilizar cualquier otro desarrollo similar, con algunos LEDs conectados al puerto B de forma que se enciendan cuando pone un 0 en la salida del SX (ver Figura II.1). Conecte cada LED en pines adyacentes del puerto B del SX. Si es muy trabajador, conecte 8 LEDs, uno a cada pin del puerto B. Puede ahorrar algo de tiempo si usa LEDs que posean resistores limitadores internos. Éstos se encuentran en varias distribuidoras y de esa forma no necesitará agregar los resistores al circuito. Simplemente conecte el LED a 5V y al pin del SX.

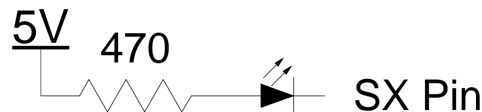

Figura II.1 – Circuito para el LED

Para comenzar, ingresará un programa en el editor del SX-Key, lo descargará en el controlador SX, y lo ejecutará. Verá el significado exacto de cada parte del programa más adelante en esta misma Unidad. Por ahora, concéntrese en familiarizarse con los pasos de la programación y la configuración del circuito.

Primer Paso

Si aún no lo ha hecho, instale el software del SX-Key como se explica en el manual. El manual también explica cómo ejecutar el programa, cosa que debería hacer en este momento. La pantalla inicial aparece en blanco, pudiendo ingresar su programa aquí (puede también, por supuesto, cargar un programa anterior desde un disco).

¿Qué escribir? ¡Ese es el problema! Por ahora, ingrese el siguiente programa simple tal como se muestra. Note que cada línea excepto la que dice **punto_de_inicio** está separada del margen por una tabulación. Es común en el lenguaje assembler poner rótulos (como **punto_de_inicio**) en la primer columna, y colocar los comandos una columna hacia la derecha, empleando el tabulador.

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


                    org    0

punto_de_inicio     mov    !rb,#0     ; hace todo el puerto b salidas
                    mov    rb,#0      ; todas las salidas del puerto b = 0
                    sleep             ; va a dormir (sleep)
```

Por defecto el software del SX-Key no distingue entre mayúsculas y minúsculas, pero puede ser configurado de esta forma si lo desea (esto hace las cosas más difíciles, así que no debería hacerlo).

Es una buena idea guardar el programa de vez en cuando. Si Windows se congela o se cuelga por cualquier motivo, estará feliz de haberlo guardado. Debería guardar su trabajo antes de intentar ejecutarlo en el SX.

Cuando termine de ingresar el programa, seleccione Run|Assemble del menú del SX-Key. Si tecleó todo el código sin errores, verá "Assembly Successful" (Ensamblado Exitosamente) en la barra de estatus. De otra forma, verá un mensaje de error y el cursor saltará a la línea que contiene el error. Corrija el error e intente otra vez.

En este punto, lo único que está haciendo el software del SX-Key es controlar los errores sintácticos de su programa. Aún es posible (y muy probable) cometer errores lógicos que el assembler no puede detectar. Piense en el funcionamiento de un corrector ortográfico de un procesador de texto. Puede avisarle si escribe 2 (dos) como "dot", pero no le avisará si lo escribe como "tos" o "los". El assembler tiene el mismo problema. Puede avisarle si ha cometido un error obvio, pero no puede decidir si su programa funciona como era de esperarse.

Ejecutar el Programa

Una vez que el programa se ensambla o compila correctamente, puede descargarlo en el chip SX. La forma más directa de hacer esto es usar Run|Program. Esto compila el programa nuevamente, si la compilación no detecta errores, carga el código de máquina en el chip SX. Puede encontrar más sobre la configuración del hardware en el Apéndice B. Por supuesto, si está usando la Placa SX-Tech, puede referirse a sus propias instrucciones de configuración de hardware y software.
Sin embargo, puede parecerle mejor el menú Run|Run. Este trabaja igual que el comando Program, pero además inicia la ejecución del programa. Si ha usado el comando Program, puede usar el comando Run nuevamente, o seleccionar Run|Clock para iniciar la ejecución.
Cualquiera sea el modo que elija deberían encenderse los LEDs conectados al puerto B. No es muy impresionante, pero es un inicio. En este momento usted sabe que su hardware funciona y su software está configurado correctamente.

Tip: Una vez programado el SX el chip retiene el programa hasta que sea reprogramado.

Si observa qué está haciendo el programa, debería preguntarse por qué se encienden los LEDs cuando los pines se ponen en cero. Esto puede resultar anti-intuitivo, pero es una práctica común. Aunque el SX puede recibir y suministrar una cantidad considerable de corriente, muchos chips pueden absorber más de la que pueden entregar. Debido a esto, los diseñadores conectan los LEDs y otras cargas de forma que se enciendan con un 0.

¿Y Ahora Qué?

Esto no parece muy impresionante. Usted puede encender LEDs sin necesidad de tanto problema, ¿no? Así que agregue la siguiente línea de código justo antes de la línea que tiene **sleep**:

```
mov   rb,#$AA   ; hace algunas salidas del puerto b = 1
```

Ahora cuando ejecute el programa, verá algunos LEDs encendidos y otros apagados. ¿Es correcto ese comportamiento? Después de todo, el programa enciende todos los LEDs al principio. Luego apaga algunos. ¿Por qué no logra ver a todos encendidos antes de que algunos se apaguen? La respuesta es que el chip SX ejecuta cada instrucción en ¡20nS! Debería tener unos ojos especiales para lograr ver los LEDs encendidos por 20nS.

Sin embargo, si pudiera ejecutar las instrucciones del SX una a la vez, podría verlo. De hecho, esto es algo que el depurador puede hacer. Antes de enviarlo al depurador, miremos qué sucede dentro de este simple programa.

Dentro del Programa

La forma más simple de comprender lo que hace este simple programa es examinarlo línea por línea. En el proceso, verá algunos conceptos claves que manejará en cada programa que escriba. Las dos primeras líneas comienzan con la directiva **device** (dispositivo). En realidad no es un comando para el SX, sino una directiva para el compilador. Los comandos tienen un valor equivalente en código de máquina. Sin embargo, las directivas no generan código de máquina, sino que solamente dan instrucciones al compilador. En este caso queremos que el compilador sepa que escribimos un programa donde debería configurar al SX para simular al PIC16C55 (un controlador anterior), y un oscilador de alta velocidad (**oscxt5**). La segunda línea le avisa al compilador que queremos usar varios modos especiales que soporta el SX. El compilador usará esta información para quemar los fusibles de configuración del SX. Estos fusibles controlan la configuración del hardware del chip y no son una parte real del programa. Normalmente, usted reemplazará la directiva **pic16c55** con **sx28l** (para dispositivos de 28 pines) o **sx18l** (para chips de 18 pines). Sin embargo, hasta que vea la Unidad 6, es mejor emplear la emulación del 16c55.

La siguiente línea contiene la directiva **reset**. Esto le informa al compilador dónde comenzará la ejecución del programa. Usted podría pensar que sería más lógico que el programa comenzara por el principio, pero verá más adelante que este no es siempre el caso. El nombre después de la directiva, **punto_de_inicio**, es un rótulo definido por el usuario. Este rótulo puede ser cualquier identificador que elija para ubicar un punto del programa.

Tip: Los rótulos y otros identificadores pueden ser de hasta 32 caracteres. El primer carácter debe ser una letra o un guión de subrayado. Los otros caracteres pueden ser letras, guiones, o dígitos. No puede usar palabras reservadas (como **sleep** y **reset**) como identificadores.

La siguiente línea especifica la frecuencia del reloj en Hertz. Esto en realidad no tiene efecto sobre el chip SX, pero ayuda al depurador comunicándole qué frecuencia de reloj quiere usar. Si no especifica una directiva **freq**, tiene un valor por defecto de 50MHz. También puede variar la frecuencia del reloj ejecutando los programas mediante el menú Run|Clock. El compilador le permite usar guiones separadores en cualquier número, con el objeto de hacerlo más legible. Así podría escribir la línea de esta forma:

```
            freq         50_000_000
```

La siguiente línea contiene la directiva final, **org** (que viene de *origen*). Esta directiva hace que el compilador comience la generación de código en una dirección en particular. En este caso, se desea comenzar desde el principio, así que **org** es 0.

Las siguientes 3 líneas (o 4 si ha agregado la línea de código que apaga algunos LEDs) son el verdadero programa. Las líneas anteriores a este punto eran simplemente directivas para el compilador. La primera línea de programa comienza con el rótulo **punto_de_inicio**. A este punto de refiere la directiva **reset**. Observe que el rótulo aparece al principio de la línea. El resto de la línea es la verdadera instrucción para el microcontrolador.

Registros

La memoria de datos del SX consta de un pequeño número de registros de un byte. Aunque hay más de 100 registros en el SX, su programa sólo puede trabajar con 32 a la vez. En una Unidad posterior, aprenderá a manejar todos los registros mediante *banking (bancos)*, pero por ahora, nos basta decir que cuenta con 32 registros. Los registros $08 a $1F están disponibles para su almacenamiento de datos. Sin embargo, los registros $00 a $07 son especiales debido a que controlan al chip SX a medida que su programa se ejecuta.

Por ejemplo, el registro $05 corresponde al puerto A del SX. Cuando lee un valor del registro $05 (conocido como el registro **ra** por el compilador), en realidad está leyendo las señales digitales presentes en los pines de entrada del puerto A. Si escribe en el registro **ra**, modificará las señales digitales que aparecen en los pines de salida del puerto A. También puede usar $06 (**rb**) o $07 (**rc**). En el dispositivo de 18 pines (que no tiene puerto C) puede usar el registro $07 para almacenar datos si así lo desea.

Esto nos lleva a otro problema: Cómo sabe qué pines son entradas y cuáles son salidas. En principio, todos los pines son configurados como entradas. Sin embargo, su programa puede cambiar esto en cualquier momento almacenando un valor especial en el registro de dirección del puerto. Para acceder al registro de dirección del puerto ponga un signo de admiración delante del nombre del registro. Al escribir un 0 en el registro de dirección de un bit, este se convierte en salida. Un 1 lo configura como entrada.

Ahora las tres líneas del programa tienen más sentido. La primer línea usa la instrucción **mov** (mover). Esta instrucción coloca un cero al registro de dirección del puerto B (**!rb**). Observe que el cero 0 tiene un caracter # delante de él. Este lo identifica como constante. Sin el #, la instrucción movería el contenido del registro 0 a **!rb**. Puede especificar la base después del #, así #$FF es una constante hexadecimal #%1011 es una constante binaria.

La segunda línea usa la misma instrucción, pero ahora el registro de destino es **rb** en lugar de **!rb**. esto escribe los datos en el puerto. Debido a que todos los pines son salidas, cada pin tendrá un nivel de 0V. Esto hace que se enciendan los LEDs.

Si agregó la línea extra de código, escribe $AA en el puerto. Esto es lo mismo que %10101010 quedando los LEDs encendidos alternadamente. La última línea, **sleep**, apaga el controlador y lo pone en modo de bajo consumo. Raramente hará esto en un programa real (por lo menos, no de esta forma). La mayoría de los microcontroladores funcionan continuamente. Más adelante, podrá querer dormir (sleep) hasta que algún evento externo lo despierte, o transcurra cierto período de tiempo, pero en este caso el controlador duerme para siempre, algo no muy útil en el caso de un Microcontrolador.

Otra cosa que puede haber notado en el programa son los comentarios. Éstos comienzan con punto y coma y se extienden hasta el final de la línea. Puede usar comentarios en cada lugar donde quiera realizar aclaraciones sobre el funcionamiento del programa. Esta es una buena idea en el caso que alguien más quiera leer su trabajo. También puede ayudarle a revisar su propio código 6 meses después de escrito, cuando tal vez no recuerde exactamente cómo funcionaba.

Tip: Otro uso de los comentarios es para quitar temporalmente una línea de su programa. Simplemente coloque un punto y coma frente a la línea que quiere "eliminar" y en caso de querer restituirla, simplemente borre el punto y coma.

Si usted fue un programador de PBasic, podría querer pensar este programa así:

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

Observe que el PBasic usa un registro de dirección al igual que el SX. Sin embargo, el significado de cada bit es el opuesto. En un programa del Basic Stamp, un bit de dirección de 0 implica que el pin es una entrada, y un 1 fija al pin como salida.
Tomando una pieza por vez, este programa no es tan complicado. Sin embargo, hay una forma mejor de comprender lo que está sucediendo: use el depurador (debugger).

Depuración Elemental

Una vez que el programa funcione, podría intentar ejecutarlo con el depurador (debugger) para observar como trabaja (asumiendo que no está usando el SX-Blitz que no soporta depuración). Esto también le dará práctica con el uso del depurador, algo que seguramente necesitará dentro de poco. Para comenzar, use el comando Run|Debug. Éste es similar al comando Run|Run pero también carga un programa de depuración especial en el chip SX. Normalmente, usted no nota la presencia de este programa. Sin embargo, debe asegurarse de dejar cierto espacio de memoria libre o el depurador no funcionará. De hecho, se debe cumplir con las siguientes pautas para hacer posible la depuración:

- No emplear reloj externo (el SX-Key suministra el reloj)
- Usar la directiva **RESET**
- No usar watchdog timer (temporizador de control, se verá más adelante)

- 2 instrucciones libres en el primer banco de memoria de programa
- 136 instrucciones libres en el último banco de memoria
- Una directiva **FREQ**, a menos que quiera ejecutarlo a 50MHz en cuyo caso **FREQ** es opcional

Después de presionar Run|Debug verá las ventanas de programación usuales. Luego se abrirán tres ventanas. La ventana Registers (Registros) muestra el contenido de los registros del SX y una muestra del código de máquina que se está ejecutando. La ventana Code (Código) muestra su código fuente (y el código de máquina a su izquierda). Finalmente, la ventana Debug (Depurar) le permite el control remoto para iniciar y detener la ejecución del programa de muchas formas distintas.

La primer cosa que observará es que su programa comienza al principio de la memoria, y no en su punto de inicio. Esto es debido a que el SX siempre se inicia en la dirección más alta. Debido a que le dijo al SX que simule a un 16c55, este programa comienza en la dirección $1FF (esto variará en otros modos, pero siempre será la dirección más alta posible). En esta dirección hay una instrucción **JMP** que el compilador escribió por usted, basado en la información de la directiva **RESET**. Esta instrucción salta a una dirección diferente determinando la dirección de comienzo de su programa.

En la Figura II.2 observará que la ventana Register tiene los primeros 16 registros del SX al costado izquierdo de la pantalla. Verá los registros **RA**, **RB**, y **RC**, así como los registros del usuario $08 a $0F. Los valores se muestran en hexadecimal, pero inmediatamente a la derecha de cada valor, se muestra el mismo número en binario. Los otros registros (en hex solamente) se encuentran al costado derecho de la ventana Register.

El centro de la pantalla muestra la porción de programa que se está ejecutando. Observe que algunas instrucciones que escribió en su programa en realidad generan más de una instrucción en lenguaje de máquina. Por ejemplo, la línea que dice:

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

En realidad genera:

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

El registro **W** (que aparece en la parte superior de la ventana de registros) es un registro especial conocido como *acumulador*. Prácticamente todas las operaciones matemáticas se producen en el registro **W**.

No hay ninguna instrucción para mover una constante al registro **!rb**, así que el compilador emplea automáticamente el registro **W**. Si esto no se tiene en cuenta puede producir errores. Por ejemplo, considere esto:

```
mov w,#$AA
mov !rb,#0
; Ahora w almacena 0, aunque usted piense que tiene $AA
```

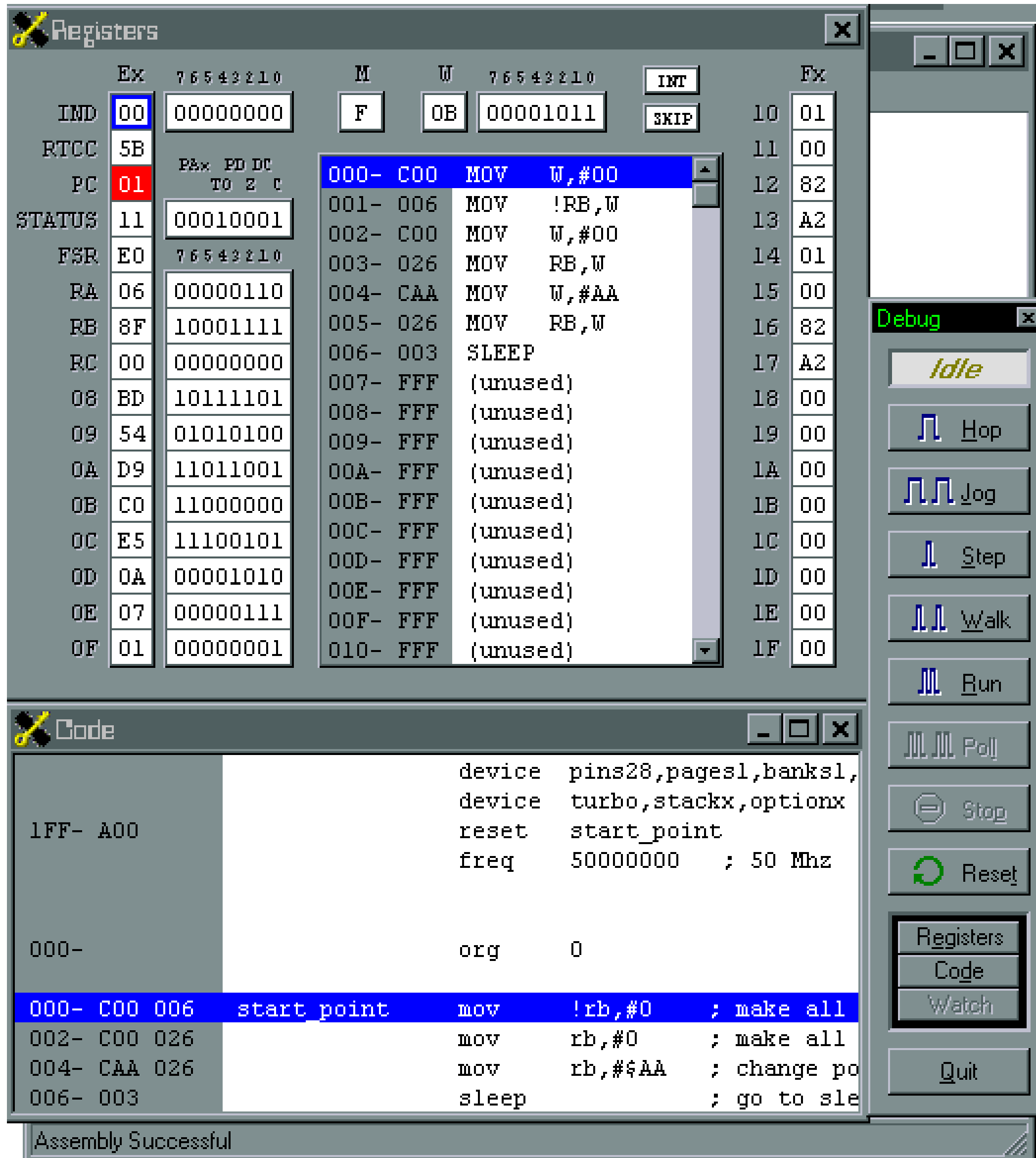

Figura II.2 – El Depurador (Debugger)

El control remoto tiene controles que puede usar para estudiar el programa:

- Hop – Ejecuta una instrucción de lenguaje assembler (recuerde, esto podría ser más de una instrucción de código de máquina)
- Jog – Ejecuta lentamente las instrucciones del assembler permitiéndole observar el funcionamiento de su programa. Presione Stop para detener el modo Jog
- Step – Ejecuta una instrucción de código de máquina
- Walk – Similar al modo Jog, pero ejecuta las instrucciones de código de máquina en lugar de las instrucciones en lenguaje assembler
- Run – Ejecuta su programa a toda velocidad. El depurador no puede examinar los registros hasta que presione Poll o Stop
- Poll – Este botón sólo se activa si el programa se está ejecutando. Hace que el compilador congele al controlador momentáneamente, lee los registros para que usted pueda verlos, y rehabilita la ejecución del programa
- Stop – Detiene los comandos Jog, Walk, o Run (solamente está activo cuando estos comandos se están ejecutando)
- Reset – Ejecuta el programa desde el principio

A medida que recorre el programa, verá una línea resaltada que le indica la instrucción que está ejecutando. Además, los registros que cambiaron su valor aparecerán en rojo.

Deteniendo el Depurador

Este es un programa corto, así que es fácil de controlar. Sin embargo, este no es siempre el caso. A menudo, el área que quiere revisar se encuentra a la mitad de un programa largo. Esa pieza de código tal vez se ejecute solamente disparada por un evento externo, o luego de una pausa. En este caso, necesitará fijar un *breakpoint*.

Un breakpoint (punto de detención) es una señal de stop (parar) en su programa. Cuando el SX intenta ejecutar la línea de código donde se encuentra el breakpoint, el depurador toma el control y el programa detiene su ejecución. Puede continuar la ejecución usando el control remoto Debug, tanto en modo normal como por pasos.

El depurador sólo soporta un breakpoint por vez. Para fijar un breakpoint, haga click en la línea que desea detenerse (en la ventana Register o en Code). La línea se pondrá roja. Ahora si presiona Run (asegúrese de presionar Reset si ya ha ejecutado el programa) la ejecución del programa de detendrá en el breakpoint. Al crear otro breakpoint, se borra el anterior. Si quiere eliminar todos los breakpoints, simplemente haga click sobre la línea que ya tiene un breakpoint.

También puede agregar un breakpoint en su programa en lenguaje assembler para obtener un breakpoint cuando inicia la depuración. Esto se logra mediante la directiva **BREAK** como a continuación:

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

Si intenta poner un **BREAK** antes de una instrucción **sleep** no funcionará. Si debe hacer esto, use una instrucción **NOP** después de break. **NOP** viene de "ninguna operación", la instrucción no hace más que perder tiempo. Debería usar este mismo truco cuando depura un programa que salta a la misma dirección usando **jmp**.

Resumen

A esta altura ya conoce cuatro instrucciones, **mov**, **sleep**, **nop**, y **jmp**. Hay mucho más que aprender sobre la instrucción **mov**, pero aún así es obvio que necesita conocer más instrucciones para escribir cualquier programa útil. Sin embargo, estas pocas instrucciones nos permiten controlar las salidas del SX. En la Unidad siguiente, aprenderá más sobre saltos y rótulos y agregará más funciones a este simple programa.

Ejercicios

1. Debido a que cada bit del registro de dirección representa a un pin distinto, tiene más sentido especificar el valor del registro de dirección (y el registro de puerto también) en números binarios. Rescriba el primer programa de ejemplo de esta unidad usando números binarios en lugar de hexadecimales.
2. La instrucción **JMP** transfiere el control a distintas direcciones. ¿Puede reemplazar la instrucción **SLEEP** con un **JMP** (salto) hacia el principio del programa? Prediga cómo afectará esto a los LEDs.
3. El problema con el programa de esta unidad es que los LEDs cambian demasiado rápido, no puede verlos sin el depurador. ¿Puede reducir la velocidad del SX para poder visualizar el cambio en los LEDs sin el depurador?

Respuestas

1. Cambie #0 por #%00000000 y #$AA por #%10101010
2. Cambie el comando **sleep** por:
   ```
   jmp punto_de_inicio
   ```
   Ahora los LEDs cambian una y otra vez rápidamente. Usted no puede notar el parpadeo en las luces, pero observará que las que parpadean brillarán menos que las fijas.
3. Usando Run|Clock, puede reducir la velocidad a 400kHz. Sin embargo, esto es aún demasiado rápido para percibir el cambio en los LEDs. Probablemente la mejor forma de ver lo que sucede en el programa es usar los comandos Jog o Walk del depurador.

Los programas y la información de este texto son para propósitos educativos, habiendo sido cuidadosamente probados, pero no están garantizados para ningún propósito en particular. El editor, el autor y el traductor no ofrecen ninguna garantía sobre la precisión, adecuación, o completitud de la información presentada, y no asumen ninguna responsabilidad por errores u omisiones en el texto. Tampoco asumen la responsabilidad por daños emergentes por el uso de la información de este texto, o por cualquier violación de los derechos intelectuales o de autor de terceros, que resulten del uso de esta información.

Versión en Castellano 1.0

3

Unidad III. Control de Flujo Simple

Unidad III de Introducción al Lenguaje Assembler con el Microcontrolador Scenix SX

© 1999 Parallax, Inc. Reservados todos los derechos. Autor Al Williams, AWC. Traducción Arístides Álvarez.

En la unidad anterior, escribió y depuró un programa simple. Este programa comenzaba en la dirección 0, ejecutaba un conjunto simple de instrucciones, y luego se iba a dormir (bajo consumo). Esto sirvió para comenzar, pero normalmente los microcontroladores no ejecutan unas pocas instrucciones y se detienen, sino que funcionan indefinidamente, revisando las entradas y controlando las salidas.

En esta unidad, extenderá el programa simple anterior de forma que realice acciones más interesantes. Al mismo tiempo, conocerá algunas instrucciones más del SX.

Contador de Programa

A medida que ejecutaba el programa de la unidad anterior en el depurador, habrá visto que el registro **PC** cambió cada vez que se ejecutó un paso (step). Si profundizó esta observación, puede haber notado que el número de **PC** correspondía a la instrucción de código de máquina en ejecución. Esto es debido a que el **PC** es el *contador de programa*. Este es un registro especial que le dice al SX qué instrucción ejecutará a continuación.

¿Recuerda la primer instrucción que vio en el depurador? Era un **JMP** puesto automáticamente por el compilador para que el programa comenzara donde nosotros pretendíamos. Por supuesto, también puede escribir sus propias instrucciones **JMP** para controlar el flujo de la ejecución de sus programas. Esto es similar al uso de la instrucción **goto** en Basic o C.

En los ejercicios de la unidad anterior, cambió la instrucción **sleep** a **JMP** para lograr que el programa se repita en vez de detenerse. Sin embargo, la solución presentada no fue la más eficiente. Esta es la solución completa:

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


                org     0

punto_de_inicio         mov     !rb,#0     ; hace puerto b salidas
                        mov     rb,#0      ; hace salidas puerto b = 0
                        mov     rb,#$AA    ; modifica salidas puerto b
                        jmp     punto_de_inicio
```

¿Qué está mal? No hay nada realmente malo en el código. Sin embargo, tal como está escrito almacena repetidamente un 0 en el registro de dirección (**!rb**). Esta es una acción innecesaria. Una vez fijado el registro de dirección, no es necesario volver a escribir el mismo valor. Esto no ocasiona ningún error, pero desperdicia tiempo que podría usar para hacer alguna otra cosa.

La solución es simple. Agregue otro rótulo a la línea siguiente a **punto_de_inicio**. Llámelo **repetir**. Usted puede entonces saltar a **repetir** en lugar de **punto_de_inicio**. Así:

```
punto_de_inicio   mov     !rb,#0    ; hace puerto b salidas
repetir           mov     rb,#0     ; hace salidas puerto b = 0
                  mov     rb,#$AA   ; modifica salidas puerto b
                  jmp     repetir
```

Otra forma de hacer el programa más legible es usar la instrucción **CLR**. La instrucción **CLR** fija a cero a cualquier registro normal o al registro **W** (no puede usarlo con el registro **!rb)**. También es más eficiente debido a que al usar **MOV** para limpiar un registro normal, se necesitan dos instrucciones, en oposición a la instrucción simple **CLR**. Este es el código:

```
punto_de_inicio   mov     !rb,#0    ; hace puerto b salidas
repetir           clr     rb        ; hace salidas puerto b = 0
                  mov     rb,#$AA   ; modifica salidas puerto b
                  jmp     repetir
```

¿Más Interesante?

Para hacer el programa más interesante, necesitará algunas instrucciones más. Considere la instrucción **INC** (incrementar). La instrucción **INC** le suma 1 a un registro. Dado que los pines del puerto B trabajan como un registro (el registro **rb**), puede incrementarlo al igual que cualquier registro.
Modifique su código para que se vea así:

```
punto_de_inicio   mov         !rb,#0    ; hace puerto b salidas
repetir           clr         rb
                  inc         rb        ; modifica salidas puerto b
                  jmp         repetir
```

¿Qué debería hacer este programa? Usted pretendería que el programa encendiera las luces con un patrón binario. Al comenzar todas las luces estarían encendidas, luego el LED del pin 0 se apagaría. Luego volvería a encenderse a la vez que se apagaría el LED del pin 1. Sería equivalente a una cuenta en binario donde los LEDs encendidos representan un 0.
Eso es lo que usted pretendería de este código, pero no funciona. Pruébelo. Cuando ejecuta el código, los LEDs parecen permanecer encendidos. Si ejecuta el código paso a paso, verá algo un poco distinto. Use el depurador para determinar qué está mal en el programa (incluso aunque ya lo haya notado) y luego lea la sección siguiente.

3

¿Qué esta mal?

Como ya debe haber notado, el problema es que al saltar a **repetir,** el programa rescribe el registro **rb** con 0. Para solucionar esto, mueva el rótulo **repetir** a la línea siguiente:

```
punto_de_inicio     mov         !rb,#0     ; hace puerto b salidas
                    clr         rb
repetir             inc         rb         ; modifica salidas puerto b
                    jmp         repetir
```

Ahora el programa trabaja como se esperaba. Si tiene un osciloscopio, puede interesarle observar los pines del puerto B. Bit 0 del puerto B generará pulsos de cierto ancho dependiendo del reloj del sistema. Bit 1 emitirá pulsos del doble de longitud. Bit 2 creará pulsos 4 veces más largos, y así. Usando los tiempos de ejecución de cada instrucción provistos por las hojas de datos del SX, puede calcular estos tiempos. La instrucción **inc** necesita 1 ciclo de reloj (20nS a 50MHz) y **jmp** requiere 3 ciclos (60nS a 50MHz). Así que el pin cambiará cada 80nS. Para fines prácticos, ha creado un oscilador de onda cuadrada y un divisor, todo en software.

No se dijo nada de que el SX puede ejecutar las instrucciones en dos modos: modo de compatibilidad, y modo turbo. En el modo de compatibilidad, el SX necesita más tiempo para ejecutar cada instrucción. Por ejemplo, en ese modo, un **inc** requiere 4 ciclos de reloj. Esto hace compatible al SX con programas escritos para microcontroladores PIC (de Microchip) que necesitan una ejecución más lenta. Todos los programas de este libro usan la cláusula **turbo** en la directiva **device**, y por lo tanto usan un cuarto del tiempo de ejecución que se emplearía en el modo de compatibilidad. En programas nuevos siempre es preferible emplear el modo turbo para obtener el mejor rendimiento.

Cuando programa microcontroladores, a menudo es necesario computar el número de instrucciones que se ejecutarán para poder determinar exactamente los tiempos. Algunas veces deberá hacer esto para definir pausas. Otras veces determinará una frecuencia, como en este caso. Cuando necesite calibrar sus tiempos, encontrará útil la instrucción **nop** (la que no hace nada). Ya vio esta instrucción en la unidad anterior. Simplemente desperdicia 1 ciclo de reloj.
En PBasic, usaría un programa similar a:

```
DIRL=%11111111          ' todas salidas
OUTL=b1
bucle:
      b1=b1+1
      OUTL=b1
GOTO bucle
```

Hay que considerar qué sucede cuando algunos de los pines del puerto B son entradas (lo que no pasa en nuestro caso). Podría ser un problema porque la instrucción incremento lee el puerto, incrementa el valor que encuentra, y luego escribe este nuevo valor en el mismo puerto. Cuando algunos pines son entradas, la instrucción leerá los pines de entrada correctamente, reflejando el estímulo externo del SX chip. Cuando incrementa eso, puede no obtener el valor esperado.

Por ejemplo, suponga que el bit 7 sea una entrada. Cuando escribe un 0 en el puerto, no tiene efecto sobre el bit 7. Si el pin 7 del puerto B tiene aplicado un nivel bajo, la primer instrucción **INC** trabajará como se esperaba. Leerá un 0 y escribirá un 1 en la salida. Sin embargo, si el pin tenía un nivel alto, la instrucción **INC** leerá un %10000000 y escribirá %10000001. Esto probablemente no será muy perjudicial, pero hay casos donde sería un gran problema. Siempre sea precavido al usar instrucciones que leen, modifican, y escriben en los puertos de E/S.

Otras Formas de JMP

La instrucción **jmp**, se puede usar de otras dos formas. En una puede usar el registro **W** (el acumulador) como dirección de destino. Simplemente escriba:

```
JMP         W
```

Esto es útil cuando desea realizar un cálculo para determinar hacia dónde saltar. La otra forma de **JMP** no se ve como **JMP**. La instrucción **ADD** le permite sumar el registro **W** al registro **PC**. Esto causa un salto sobre cierto número de instrucciones. Por supuesto, la instrucción **ADD** simplemente suma el registro **W** con cualquier otro registro. Lo que ocurre es que al modificar el valor de **PC** se realiza un salto. Por ejemplo, considere esto:

```
CLR         8      ; limpia registro 8
MOV         W,#2
ADD         PC,W
INC         8
INC         8
INC         8
BREAK              ; ¿Qué hay en el registro 8 ahora?
INC         8
```

Cuando el depurador llega al breakpoint, el registro 8 contiene un 1 debido a que al modificar **PC** no se ejecutan las dos primeras instrucciones **INC**. El compilador le permite escribir esta instrucción como **JMP PC+W** para hacer el programa más simple de leer.

Tip: Debido a que el 2 en este ejemplo es una constante, podría usar una instrucción **JMP** común para saltar sobre las dos instrucciones. Una forma, por supuesto, es colocar un rótulo hacia donde saltar. Sin embargo, también puede usar el rótulo especial **$,** que simboliza la dirección actual. Así que se podría escribir **jmp $+3**. Por qué +3 en vez de +2? Debido a que $ se refiere a la dirección actual, debiéndole agregar 1 para ejecutar la instrucción siguiente. Usando +2 solamente saltaría 1 instrucción.

La verdadera utilidad de usar **ADD** para realizar un salto es cuando computa un ajuste durante la ejecución. Esto le permite crear datos y moverse sobre tablas como verá más adelante en este libro.

En este ejemplo, usar un 8 como número de registro es confuso. Recuerde, no es una constante debido a que no comienza con #. Sin embargo, es mucho más prolijo llamar a sus variables de un modo más significativo. El compilador le proporciona un par de formas de hacer esto como se verá en la próxima unidad.

Por supuesto, algunas veces desea saltar solamente si una condición es verdadera o falsa. Por ejemplo, puede querer saltar solamente si el usuario presiona un botón, o cuando un sensor lee cierto valor. Aprenderá todo esto en la Unidad V.

Rótulos Locales

Un desafío cuando se encuentra programando, es encontrar nombres nuevos para cada rótulo. El compilador del SX-Key le permite crear *rótulos locales* que comienzan con dos puntos. Estos rótulos solamente son válidos entre rótulos globales. Debido a que los rótulos locales son válidos entre rótulos globales, puede definir el mismo rótulo más de una vez sin confusión. Considere este ejemplo:

```
inicio      mov         w,#0              ; rótulo global
:bucle     .                              ; primer bucle
           .
           .
            jmp         :bucle            ; va al primer bucle
ejemplo     mov         w,#9              ; rótulo global
:bucle     .                              ; segundo bucle
           .
           .
            jmp         :bucle            ; salta al segundo bucle
```

Nunca ha usado rótulos locales. Sin embargo, usarlos puede simplificarle las cosas y hacer su programa más legible. La alternativa es generar rótulos únicos para cada dirección de interés en su programa.

Otra Forma de INC

Algunas veces quiere incrementar el valor de un registro, sin alterar el valor que se encuentra almacenado en ese registro. En este caso puede usar una forma especial de la instrucción **mov**:

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

Esta instrucción entrega el resultado en el registro **W** sin modificar el registro 8. Esto le permite usar el valor almacenado en el registro en varios cálculos sin modificarlo.

En general, las operaciones matemáticas funcionan de esa forma. Por ejemplo, lo opuesto de incrementar es disminuir (**dec**). Esta función le resta uno al valor del registro. Puede escribirla como:

```
dec 8
```

o:

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

La primera forma le resta uno al valor del registro 8 actualizando el valor. La segunda forma realiza la resta entregando el resultado en **W** sin modificar el valor original.

Tip: Basic no tiene una analogía exacta a **inc** y **dec** (otra que no sea **x=x+1** o **x=x-1**). Sin embargo, en C, puede pensar a **inc** y **dec** como los operadores ++ y –-, respectivamente.

Deteniendo el Controlador

En los ejemplos anteriores, el programa usó la instrucción **sleep** para detener el controlador. Esto puede parecer no muy práctico, pero en algunos casos será de utilidad. Por ejemplo, imagine un microcontrolador que marque un número telefónico de emergencia. La señal de inicio podría ser la aplicación de alimentación al circuito. El programa marcará el número y se irá a dormir, esperando otro ciclo de alimentación para ejecutarse nuevamente.

Sin embargo, la razón principal por la que usará la instrucción **sleep** es poner el controlador en modo de bajo consumo hasta que ocurra algún evento externo o pase cierto período de tiempo. Los eventos externos normalmente toman la forma de interrupciones, tema que trataremos en la Unidad VII. Sin embargo, puede reactivarse en un tiempo predeterminado mediante el temporizador watchdog. El propósito principal del temporizador watchdog es reiniciar el controlador en el caso de una falla. Sin embargo, también puede usarlo como temporizador para fijar el tiempo de reactivación.

Sobre el Watchdog

Para habilitar el watchdog, agregue la configuración de **watchdog** a la directiva **device** al comienzo del programa. Observe que al habilitar el watchdog el depurador no trabajará correctamente. La idea detrás del watchdog es que su programa debería usar la instrucción **clr** para fijar a cero el registro **!WDT** periódicamente. Esto indica que el programa está funcionando. Si falla en limpiar este registro durante un cierto período de tiempo, el controlador se reinicia.

Tip: El propósito original del temporizador watchdog es reiniciar el controlador en caso de falla. Normalmente es mejor tener un solo punto del programa en el que se limpia el registro del watchdog (**WDT**). De esa forma las posibilidades de que su programa falle y continúe limpiando el temporizador son remotas. Si su programa deja de comportarse correctamente, el temporizador watchdog lo reiniciará.

¿Cuánto dura este período? El SX tiene un oscilador interno para el watchdog que normalmente funciona a 14kHz y el watchdog cuenta 256 pulsos. De esta forma el período máximo de control es de 18mS. Así, si no ejecuta una instrucción **CLR !WDT** por lo menos una vez cada 18mS, tendrá un reinicio por watchdog.

Para aplicaciones de temporización, este no será un período suficientemente largo. El SX le permite multiplicar el tiempo del watchdog configurando el registro **!OPTION**. En particular, en el bit 3 de ese registro va un 1 si quiere usar el prescaler con el temporizador watchdog. Bits 2, 1, y 0 fijan el valor de división (ver tabla III.1). El valor máximo es 1:128 dando un período de tiempo de 2,3 segundos.

Bit 2	Bit 1	Bit 0	Factor
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

Tabla III.1 – Valores de Escala del Temporizador Watchdog

¿Cómo fijar el valor de un bit de un registro? Puede usar **SETB** para fijar un bit a 1 y **CLRB** para fijar el bit a 0. Para activar el prescaler del watchdog y fijar el factor en 1:32 escribiría:

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

La ventaja de hacerlo de esta forma es que no perturba el resto del registro. Sin embargo, también se puede observar que el valor por defecto del resto de los bits del registro **!option** son unos. Entonces sabiendo que necesita unos en el resto del registro, podría escribir:

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

El registro **!option** tiene por defecto todos sus bits igual a uno, así que si busca el máximo valor de temporización, no necesita hacer más que habilitar el temporizador watchdog. Considere este programa:

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

                org     0

punto_de_inicio mov     !rb,#0     ; hace salidas puerto B
agn             mov     w,#$FF
                xor     8,w        ; invierte bits
                mov     rb,8
                sleep
```

Este programa hace titilar los LEDs de forma que pueda verlos. El único problema es que no puede saber qué LEDs estarán encendidos y cuáles apagados inicialmente. El programa usa la instrucción **xor** para la operación lógica de o-exclusiva entre el registro 8 y la constante $FF. Leerá más sobre **xor** en la próxima unidad, pero por ahora tenga en cuenta que estas instrucciones invierten los bits del registro 8. Esto quiere decir que cada 0 del registro 8 se hará 1 y cada 1 de hará 0. Usted puede reemplazar las instrucciones **mov** y **xor** con la instrucción **not** que también invierte los bits y tarda menos en ejecutarse. Por ahora, sin embargo, deje el código como está, debido a que usaremos la constante $FF en la próxima unidad para demostrar algunas ideas importantes.

Lo último que hace el programa es almacenar el contenido del registro 8 en el puerto B. Dado que el código invirtió los bits, los LEDs que estaban encendidos se apagarán y los que estaban apagados se encenderán. Luego el SX se va a dormir. Sin embargo, debido a que el watchdog está encendido (observe la cláusula **watchdog** de la segunda línea **device**) el controlador se reiniciará en aproximadamente 2,3 segundos. Esto invertirá los bits del registro 8 nuevamente, cambiando el estado de los LEDs. No olvide que no puede depurar este programa debido a que usa el watchdog. Tendrá que usar el comando Run | Run para ver funcionar el programa.

Antes dijimos que los programas que usan el watchdog deben tener **clr !wdt** para reiniciar el temporizador. Este programa, sin embargo, no reinicia el watchdog. ¿Por qué? Debido a que este programa deliberadamente busca que el temporizador watchdog lo reinicie. Esta es la forma en la que se obtiene la pausa para poder observar el parpadeo de los LEDs.

Por supuesto, sería bueno saber si es efectivamente el watchdog quien reinicia el programa. Puede hacer esto examinando los bits del registro **status**. En particular, el bit 4 será 0 si el watchdog originó un reinicio. Si el bit 3 es 0, entonces la instrucción **sleep** estaba activa en ese momento. Si supo controlar estos bits (tema que trataremos en breve) podría inicializar el registro 8 en un valor conocido cuando un reinicio real suceda y no inicializarlo cuando ocurra un reinicio por watchdog.

Usar el watchdog como temporizador es un poco inusual, pero perfectamente legítimo. En las próximas unidades encontrará otras dos formas de crear pausas: programando bucles y usando el reloj de tiempo real. Estos serán más simples debido a que podrá usar el depurador con estos métodos. Otra ventaja: cuando el controlador se reinicia, hay un pequeño intervalo de tiempo en el que todos los pines se configuran como entradas hasta que el programa modifica el registro de dirección. Los otros métodos de pausas le permitirán a su programa mantener el control sobre el procesador todo el tiempo.

Resumen

Esta unidad cubre muchas instrucciones que incluyen:

- **jmp** – salta hacia otro punto del programa
- **sleep** – detiene el controlador
- **inc** – suma 1 a un registro (también use **mov w, ++r** para almacenar el resultado en **w**)
- **dec** – resta 1 a un registro (o use **mov w, --r**)
- **nop** – no hace nada durante 1 ciclo de reloj
- **setb** – pone un 1 en un bit de un registro
- **clrb** – pone un 0 en un bit de un registro

- **clr** – borra el contenido de un registro (**w** o el watchdog por ejemplo)
- **not** – invierte los bits de un registro
- **xor** – o exclusiva entre los bits de dos registros (más en la siguiente unidad)
- **add** – suma **w** a un registro (más en la siguiente unidad)

También leyó sobre el registro **PC**, y partes de los registros **!option** y **status**. En la siguiente unidad, encontrará más sobre aritmética y variables, allanando el camino hacia programas más poderosos.

Ejercicios

1. Si tiene acceso a un osciloscopio, agregue algunas instrucciones **nop** a los programas que hacen titilar los LEDs y examine los resultados.
2. Modifique el programa con watchdog de forma que los LEDs titilen a la mitad de la velocidad original (aproximadamente 1,15 segundos).
3. ¿Cómo haría para detener el programa del watchdog sin usar **sleep** ni reiniciar por watchdog? Modifique el código para que se detenga y no se reinicie. Esto resultará en un patrón estático de LEDs encendidos y apagados.

Respuestas

1. Este es un ejemplo de la solución:
```
Punto_de_inicio mov     !rb,#0     ; puerto b salidas
                clr     rb
repite          inc     rb         ; cambia salidas puerto b
                nop                ; agregue más nops si quiere
                jmp     repite
```
2. Para modificar la velocidad de parpadeo, cambiará el valor del prescaler del watchdog. Una forma de lograrlo es colocar **mov !option, #$FC** cerca del inicio del programa. Puede también usar **setb** y **clrb** para fijar individualmente los bits del registro **!option**.
3. Reemplace la instrucción **sleep** con:
```
detener clr !wdt
        jmp detener
```

Los programas y la información de este texto son para propósitos educativos, habiendo sido cuidadosamente probados, pero no están garantizados para ningún propósito en particular. El editor, el autor y el traductor no ofrecen ninguna garantía sobre la precisión, adecuación, o completitud de la información presentada, y no asumen ninguna responsabilidad por errores u omisiones en el texto. Tampoco asumen la responsabilidad por daños emergentes por el uso de la información de este texto, o por cualquier violación de los derechos intelectuales o de autor de terceros, que resulten del uso de esta información.

Versión en Castellano 1.0

Unidad IV. Variables y Matemática

Unidad IV de Introducción al Lenguaje Assembler con el Microcontrolador Scenix SX

© 1999 Parallax, Inc. Reservados todos los derechos. Autor Al Williams, AWC. Traducción Arístides Álvarez.

El SX usa sus registros para almacenar datos. En los ejemplos de las unidades anteriores, nos referimos a los registros simplemente por sus números. Recuerde, los primeros siete u ocho registros (dependiendo del tipo exacto de procesador) tienen nombres y funciones especiales (como **rb**, **status**, u **!option**).

Los nombres especiales de estos registros le ayudan a recordar la función que desempeñan. ¿Cómo puede darle nombres significativos a los registros que usa?

Suponga que quiera usar el registro 8 como una variable en su programa. Hay varias formas para hacerlo. Primero, puede escribir una ecuación de dos formas distintas. Cerca del inicio del programa podría escribir:

```
Mivar EQU 8
```

O:

```
Mivar = 8
```

Ahora puede reemplazar todos los 8 por **Mivar**. Puede usar este método para definir cualquier constante aunque no sea un número de registro. El compilador simplemente reemplaza cada ocurrencia de **Mivar** por 8.

La otra forma de definir una variable es reservándole un espacio con la directiva **DS**. La directiva **DS** normalmente se escribe después de un rótulo, y a su derecha se fija la cantidad de bytes a reservar. Así que para reemplazar las ecuaciones anteriores con la directiva **DS** podría escribir:

```
       org 8
Mivar ds 1
```

La parte confusa de esto es que la directiva **org** puede referirse al espacio de datos o al de programa, dependiendo del contexto. En este caso, el 8 se refiere a la memoria de datos. Antes de comenzar a escribir las líneas de su programa, escribirá otra directiva **org** para fijar el comienzo de su programa (a menudo ubicación 0).
Es perfectamente normal especificar varias variables una después de otra. Por ejemplo, considere este código que declara una variable de un byte llamada **Ubyte** y otra de dos bytes llamada **Dbytes**:

```
      org 8
Ubyte  ds 1
Dbytes ds 2
```

Cuando usa un nombre de variable en su programa, el nombre de una variable de varios bytes se refiere al primer byte de la variable. Considere esta instrucción:

```
	mov w,Dbytes
```

Esta carga el primer byte de la variable en w. Por otro lado, mire esta línea:

```
	mov w, Dbytes+1
```

Esta línea de código accederá al segundo byte. ¿Hay alguna diferencia con el siguiente segmento de programa?

```
      org 8
Ubyte  ds 1
Dbytes ds 1
Dbyte1 ds 1
```

No. No hay ninguna diferencia excepto que de esta forma puede usar **Tbyte1** en lugar de **Tbytes+1**. Por supuesto, puede seguir usando **Tbytes+1**; al compilador no le importa.

Un Ejemplo

¿Recuerda los programas de la unidad anterior? Este, por ejemplo:

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

                org     0

punto_de_inicio mov     !rb,#0    ; todo puerto b salidas
agn             mov     w,#$FF
                xor     8,w
                mov     rb,8
                sleep
```

Este es el mismo programa, usando nombres de variables simbólicos:

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

```
                org     8                   ; comienzo datos
valsal          ds      1

ledport         =       rb
invertir        equ     $FF

                org     0                   ; comienzo código

punto_de_inicio mov     !ledport,#0         ; hace salida puerto b
                                            ; cambio para usar una
                                            ; instrucción con xor y
                                            ; una constante
agn             xor     valsal,#invertir
                mov     ledport, valsal
                sleep
```

Observe que puede usar ecuaciones para redefinir símbolos estándar. Este programa usa = y **EQU**. Esto, a menudo, es cuestión de preferencias personales. Sin embargo, una vez que define un símbolo con **EQU** no puede ser modificado. Definir un símbolo con = le permite cambiarlo posteriormente. En este programa, como en la mayoría de los programas simples, los valores de los símbolos no cambian para nada, así que puede usar cualquier método.

Tip: Cuando define un símbolo para una constante (como **invertir**) sigue siendo necesario el caracter # delante del valor. Sin el, el compilador pensará que está definiendo un número de registro.

Otra forma de usar una ecuación es definir un nombre para un bit en particular. Puede especificar los bits en el lenguaje assembler del SX usando una extensión después del nombre del registro indicando la posición del bit. Por ejemplo, el bit menos significativo del registro **rb** es **rb.0**. El bit más significativo es **rb.7**. Usando una ecuación puede definir nombres más representativos para los bits:

```
LEDpin            equ   rb.0
```

Usar nombres para los registros y las constantes hace los programas mucho más legibles. También le permite modificar las cosas fácilmente cuando lo desee. Por ejemplo, sería muy simple modificar este programa para hacer parpadear los LEDs en el puerto A en lugar del B. Tampoco se presentarían problemas al cambiar del registro 8 a cualquier otro, si lo desea.

Asignación

En Basic o C, puede asignarle una variable a otra. El SX también puede hacer esto mediante la instrucción **mov**. Por ejemplo:

```
      org   8
byte1 ds    1
byte2 ds    1

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

Este segmento de programa pondrá $AA en **byte1** y luego el contenido de **byte1** en **byte2**.

Tip: El código de máquina del SX no tiene realmente una instrucción que mueva un registro a otro. Esto significa que el compilador genera dos instrucciones debidas al segundo **mov** del programa. Las verdaderas instrucciones son:

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

Esto quiere decir que esta línea de código modificará el registro **w**. Esto puede originar errores. Por ejemplo, considere esto:

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

Este código carga innecesariamente el registro **w** dos veces. Una forma mejor de hacerlo sería:

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

O:

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

Ambas originan 3 instrucciones (en vez de 4) y se ejecutan más rápidamente que el primer ejemplo.

El único problema son las variables de más de un byte. El SX solamente trabaja con bytes. Esto significa que si quiere trabajar con cantidades mayores, deberá descomponer las operaciones byte por byte. Por ejemplo, necesitará dos instrucciones **mov** para copiar una variable de dos bytes en otra.

Por ahora, trabajemos con bytes. Sin embargo, los bytes sólo pueden acumular valores entre 0 y 255 (o –128 a 127). Así que si necesita números mayores, no tendrá otra opción que trabajar con variables más grandes.

Mejorando la Matemática

En la unidad anterior, vio que la instrucción **add** puede sumar el registro **w** con otro. Puede dejar el resultado en **w** o en el registro que desee. También puede sumarle un número a un registro, o dos registros juntos. Sin embargo, estas son secuencias de dos instrucciones que destruyen el valor del registro **w** en el proceso. Estos son algunos ejemplos:

```
      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 destruido)
      add avar,bvar       ; avar=avar+bvar (w destruido)
      add bvar,avar       ; bvar=avar+bvar (w destruido)
```

El tamaño de un byte de estas operaciones puede causar problemas. ¿Qué sucede si la respuesta es mayor que el máximo para 8 bits? Por ejemplo, si **w** contiene $FF y le suma un registro que contiene $10, ¿qué sucede? La respuesta es que el SX trunca el resultado. Sin embargo, para hacerle saber que esto ha sucedido, activa el indicador de acarreo (carry flag), que es el bit 0 del registro **status** (estado). Esto se cumple sin importar el destino que se le de a la respuesta. Otro bit del registro **status** (bit 2) se pone en uno cada vez que la respuesta es cero. Puede usar **status.0** y **status.2** para referirse a los indicadores de acarreo y cero, o puede usar los nombres simbólicos, **C** y **Z**.

Más adelante, en esta unidad, aprenderá a examinar estos bits y emplearlos en cuentas con números con más de un byte. Debe saber que no todas las operaciones afectan a estos bits del mismo modo. Por ejemplo, las instrucciones **inc** y **dec** (tratadas en la última unidad) suman o restan 1 de un registro. Sin embargo, no activan la indicación de acarreo. Sí lo hacen, sin embargo, con el indicador de cero. Las hojas de datos del SX le informan como afectan las instrucciones al registro **status**.

Lo opuesto de sumar, por supuesto, es restar. La instrucción **sub** puede restar **w** de cualquier registro. El resultado queda en el registro. Si quiere poner el resultado en **w**, puede usar esta forma de la instrucción **mov** (donde **R** es el registro que desea usar):

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

También puede restar dos registros o un número de un registro. Sin embargo, ambas formas generan dos instrucciones en código de máquina y destruyen **W**. Así:

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

El indicador o bandera de acarreo (bit 0 de **status**) tiene el significado inverso en **sub**. Suponga que resta 100 de 30 (30-100). La bandera de acarreo estará en cero para indicar que la resta se desbordó (*underflowed)*. Sin embargo, si resta 30 de 100 (100-30), el acarreo valdrá uno para indicar que la resta se realizó correctamente. La resta también afecta la bandera de cero.

Si puede sumar y restar, podría preguntarse qué sucede con la multiplicación y la división. Los microcontroladores simples como el SX solamente pueden sumar y restar. Sin embargo, usando algunas técnicas que verá en la próxima unidad, podrá descomponer la multiplicación y la división en múltiples sumas y restas.

Números en Complemento a Dos

Si el indicador de acarreo está en cero después de una resta, ¿significa que la respuesta es incorrecta? No necesariamente. Cualquier microcontrolador, incluido el SX, puede manejar números negativos usando *aritmética en complemento a dos*. La idea es simple. Trate al bit más alto (bit 7, en este caso) como bit de signo. Si el bit es 0, entonces el número es positivo. Si el bit es 1, entonces el número es negativo. Para representar un número negativo, inviértalo y súmele 1. Obviamente, para ver el número original, deberá restarle 1, e invertirlo nuevamente.

Considere qué sucede si resta 60 de 40. La respuesta correcta, por supuesto, es -20. En el SX, sin embargo, se obtiene %11101100 ($EC). Si invierte este número (%00010011) y le suma 1 (%00010100) encontrará que el resultado es en realidad 20. Usted también puede escribir número negativos. Suponga que quiere sumar –5 y 10. Primero, encuentre la representación binaria de 5 (%00000101) e inviértala (%11111010). Luego súmele 1 para obtener %11111011 ($FB o 251). Si suma 10 y 251, obtendrá 261. Pero el SX no obtiene 261. Trunca el resultado a 5 (los primeros 8 bits de $105). Por supuesto, 10 + -5 es 5, así que la respuesta es correcta.

Estas operaciones son fáciles de realizar en el SX. La instrucción **not** invertirá los bits e **inc** o **dec** sumarán o restarán 1. De esta forma no es muy difícil manejar números negativos incluso durante la ejecución.

¿Cuál es el lado negativo del complemento a dos? Limita los números que puede representar. Para un byte, los números entre 0 y $7F representan 0 a 127 y los números de $80 a $FF representan –128 a –1.

Más Trucos de Acarreo

Suponga que necesita números más grandes, digamos 0 a 999. Necesitará usar más de 1 byte. En dos bytes se puede almacenar un número de 0 a 65535, suficiente para este trabajo. El problema es cómo realizar cálculos con estos números grandes.

Las instrucciones **addb** y **subb** sumarán o restarán un bit (que podría ser el bit de acarreo) a un registro. Considere este programa simple:

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

                org           8         ; inicio de datos
contador        ds            2


                org           0         ; inicio de código

inicio          clr           contador
                clr           contador+1  ; limpia ambos bytes
repetir
                                          ; hace una suma de 16-bit
                add           contador,#1
                addb          contador+1,status.0
                jmp           repetir
```

En este programa se le suma 1 a la variable de 16-bit **contador**. También se le suma el bit de acarreo a los 8 bits superiores del contador. Debido a que el bit de acarreo se activa cuando el contador se desborda, la cuenta será correcta. Puede hacer lo mismo con la sustracción usando **subb** en lugar de **addb**.

Usando más registros e instrucciones **addb** o **subb**, puede manejar números de cualquier tamaño. Un número de 24 bits (3 bytes) puede almacenar hasta aproximadamente 16 millones. Un número de 32 bits (el mismo tamaño que usan las PC Pentium; 4 bytes) puede almacenar números de aproximadamente 4 mil millones.

¡Pruébelo!

Ingrese el programa anterior y revíselo con el depurador en modo step. Rápidamente se aburrirá de ver aumentar el valor del registro 8. El comando Jog le ayudará, pero todavía se demora mucho en llegar a la parte interesante del código. Es el momento indicado para aprender otras características del debugger. Primero puede hacer click en el recuadro del registro 8 y cambiar su valor. Así, si ingresa $FE (o %11111110) en el recuadro del registro 8, estará mucho más cerca de ver el momento del desborde. Esto funciona para todos los registros visibles en el depurador.

Otra cosa confusa que debe saber, es que la variable contador se encuentra en los registros 8 y 9. Una forma más simple de observar el contenido de la memoria es emplear la directiva **watch**. Esta es una directiva para el compilador que se incluye en el programa con el objeto de mostrar un sector de la memoria con un nombre y un formato específicos. Usted especifica el lugar de la memoria, el tamaño de la variable, y el formato que desea. Para este programa, agregue esta línea en algún punto de su programa:

```
watch contador,16,UDEC
```

Esto mostrará la variable de 16-bit en el lugar denominado **contador** como un número decimal sin signo (positivo). Puede ver la lista de todos los formatos de código en la Tabla IV.1.

Código de Formato	Apariencia
UDEC	Decimal positivo (Unsigned decimal)
SDEC	Decimal con signo (Signed decimal)
UHEX	Hexadecimal positivo (Unsigned hex)
SHEX	Hexadecimal con signo (Signed hex)
UBIN	Binario positivo (Unsigned binary)
SBIN	Binario con signo (Signed binary)
PSTR	Cadena de caracteres ASCII de longitud fija
ZSTR	Cadena de caracteres ASCII terminada en cero

Tabla IV.2 – Códigos de Formato Watch

Tip: ASCII (American Standard Code for Information Interchange = Código Estándar Americano para Intercambio de Información) es una forma de representar caracteres de texto como números de 7 u 8 bits. Por ejemplo, en ASCII, la A es $41, un espacio es $20, etc.

Algunas Funciones Más

A menudo usará el acarreo con distintas funciones. Antes, en este libro, leyó que podía usar **setb** y **clrb** para modificar el valor de un bit. Debido a que el bit de acarreo es simplemente un bit del registro **status**, puede emplear esas instrucciones para modificar su valor.

Sin embargo, esta es una tarea realizada frecuentemente, así que el compilador provee otras instrucciones para que escriba menos. En particular, **clc** limpia el acarreo (**clz** limpia la bandera de cero) y **stc** activa el acarreo (**stz** lo hace con la bandera de cero).

La utilidad de esto es controlar el flujo de su programa basándose en estas banderas o indicadores. Hay varias formas de hacer esto. La primera de todas, la instrucción genérica **jb** realizará un salto si el bit especificado está activado (vale 1). Así que para saltar a **etiq1** si la bandera de acarreo vale 1, podría escribir:

```
jb status.0,etiq1
```

Por supuesto, usando **jb** puede especificar cualquier bit. Sin embargo, los bits de acarreo y de cero se controlan muy a menudo, así que el compilador le permite usar las instrucciones **jc** y **jz** en estos casos. También puede usar **jnb** (o **jnc** o **jnz**) para saltar cuando el bit es cero en lugar de uno.

Al hacer una resta y controlar los valores de las banderas de cero y acarreo, puede escribir fácilmente programas que decidan si un número es mayor, igual, o menor que otro número. Por ejemplo, suponga que quiere saber si la variable **x** es mayor que la variable **y**:

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

Esto funciona debido a que al restar **x** de **y** solamente será negativo (causando un desborde) si **x** es mayor que **y**. Recuerde que el acarreo es cero en caso de desborde en una resta.

Tip: Podría considerar calcular **x-y** cambiando **jnc** por **jc**. Esto también funcionaría, pero cuando **x** fuera mayor o igual que **y**. Para ver por qué, piense en el caso que **x** sea igual a **y**. Por supuesto, puede usar **jz** para controlar una igualdad y **jnz** para desigualdades. Vea una lista de resultados posibles cuando resta dos números en la Tabla IV.2.

Acarreo	Cero	Significado
0	0	a<b
X (sin importancia)	1	a=b
1	0	a>=b

Tabla IV.3 – Resultados Cuando Calcula A-B

Controlar la igualdad a cero es una operación muy común, así que el compilador le permite escribirlo en una forma especial. Puede usar **test** (probar). La instrucción **test** determina el valor de la bandera de cero, basándose en cualquier registro (incluyendo el registro **w**).

Otra función común que se aplica en la prueba de cero, se emplea al incrementar o disminuir un registro, saltando si el resultado es igual a cero. Puede usar **djnz** (para disminuir) o **ijnz** (para incrementar) con este propósito.
Este es otro programa para hacer titilar los LEDs, que usa **djnz** para repetir esto diez veces:

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

                  org   8         ; inicio de datos
contador          ds    1
patron            ds    1
```

```
                    watch contador,8,udec
                    watch patron,8,ubin

                    org   0                 ; inicio de código
inicio              mov   !rb,0             ; todas salidas
                    clr   rb                ; todas bajas
                    mov   contador,#10      ; 10 veces
repetir
                    mov   rb,patron
                    not   patron
                    djnz  contador,repetir
                    sleep
```

Observe que el parpadeo se ejecuta 10 veces debido a que la variable **contador** se inicia en 10, y se reduce 1 unidad en cada ejecución hasta que llega a cero. Esta es una idea poderosa y se usa a menudo en los programas. Este código se conoce como *bucle* debido a que se ejecuta varias veces.

En Basic o C, lograría esto con la instrucción **for**. En Basic, por ejemplo, el bucle sería así:

```
FOR contador = 10 to 1 step –1
                              ' realizar el trabajo
NEXT
```

Por supuesto, normalmente se ve en sentido inverso, con **contador** variando de 1 a 10. También se podría hacer así en assembler pero llevaría más instrucciones:

```
inc contador    ; se asume que el contador fue iniciado en 0
mov w,#10
sub w,contador-w
jnz repetir
```

En la siguiente unidad verá una serie de instrucciones de comparación que pueden realizar esta lógica en una instrucción de lenguaje assembler (pero el compilador escribirá la misma cantidad de código de máquina).

Pausas Programadas

Otro empleo importante de los bucles es el desarrollo de pausas programadas. En la unidad anterior, vio cómo usar el watchdog como un burdo sistema temporizador. Sin embargo, esa no era la forma ideal de generar una pausa. El watchdog hace que su programa sea difícil de depurar debido a que el SX-Key no puede usar el debugger cuando éste se activa. Además, el watchdog no puede generar pausas arbitrarias, y usted pierde el control del programa mientras ésta se ejecuta.

Sin embargo, si conoce la velocidad del reloj (clock), y la cantidad de ciclos que demanda cada instrucción, puede calcular la cantidad de bucles que causarán una pausa determinada. Por ejemplo, suponga que desea generar un tono de 1kHz. Un tono de 1kHz se repite cada 1mS (1/1000 = .001) así que para hacer una onda cuadrada de 1kHz, el SX debe encender un pin, esperar 500uS (la mitad de 1mS), apagar el pin, esperar otros 500uS, y repetir desde el principio.

Suponga que tiene conectado un parlante piezoeléctrico al pin 7 del puerto B (un parlante piezoeléctrico debido a su alta impedancia puede ser conectado directamente al pin de salida del SX). Si puede conmutar el pin 7 a esta velocidad, escuchará un tono de 1kHz saliendo del parlante piezoeléctrico.

El problema es que 500uS es una eternidad para el SX. A 50MHz, cada ciclo de instrucción (en modo turbo) toma 20nS. Así que para hacer una pausa de 500uS necesitará 25000 ciclos de instrucción. Considere este bucle simple:

```
      clr   pausa
bucle djnz  pausa,bucle
```

Estudiando las hojas de datos del SX, encontrará que la instrucción **djnz** toma 4 cada vez que debe saltar, y 2 ciclos si no tiene que saltar. La instrucción **clr** toma 1 ciclo. Así el número total de ciclos de este bucle es: 256 * 4 + 3 ó 1027, bastante lejos de los 25000 que usted necesita. Por supuesto, podría usar una pausa de 16-bit, pero es difícil de calcular debido a que el tiempo total del bucle varía en función del estado de la bandera de acarreo. En lugar de eso, normalmente es más simple colocar un bucle dentro de otro. Dividiendo 25000 por 1027 encontrará que necesita aproximadamente 24 repeticiones de este bucle para obtener 25000. Así:

```
            mov   pausa1,#24
bucle1      clr   pausa
bucle2      djnz  pausa,bucle2
            djnz  pausa1,bucle1
```

Por supuesto 24 * 1027 = 24648, no es la respuesta exacta. Sin embargo, el bucle exterior agrega 95 ciclos al bucle total (vea si puede calcular ese número). Se obtiene así una pausa total de 24743 (un error del 1.02%). En la mayoría de los casos, este no es un problema. Si necesita un número más exacto, podría reducir el número de ciclos del bucle interno e incrementar la cantidad del exterior hasta obtener la aproximación deseada. También puede calibrar el tiempo de los bucles empleando instrucciones **nop** en su interior.

Funciones Lógicas

Debido a que los microcontroladores trabajan con número binarios, no es sorprendente que contengan muchas operaciones para trabajar con los bits de una palabra. Como otras operaciones, estas emplean el registro **w** y otro registro arbitrario, almacenándose el resultado en el que usted elija. También puede usar un registro y una constante, o dos registros, pero si lo hace, generará más de una instrucción de código de máquina y destruirá el registro **w** en el proceso. Las principales funciones lógicas incluyen **and**, **or**, y **xor**.

¿Qué hacen estas funciones? Simplemente examinan los dos valores suministrados bit por bit y generan el bit de salida correspondiente. Tome **and** por ejemplo. Si usa **and** entre %10101010 y %11110000, el resultado es %10100000. ¿Por qué? Debido a que **and** solo deja un 1 si ambos bits son 1. La instrucción **or** pone un 1 si cualquier bit es 1. La instrucción **xor** pone un 1 si una de las entradas es 1, pero no ambas a la vez. Puede encontrar un resumen de estas operaciones en la Tabla IV.3.

Instrucción	Tabla de Verdad			Formato Mover a W
	Entrada	Entrada	Salida	
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 (or exclusiva)	0	0	0	xor w,R
	0	1	1	
	1	0	1	
	1	1	0	
Not	0		1	mov w,/R
	1		0	
RL (rotar a la izquierda)	n/a			mov w,<<R
RR (rotar a la derecha)	n/a			mov w,>>R

Tabla IV.4 – Instrucciones Lógicas

Ya ha visto que puede usar **not** para invertir los bits de un registro (incluyendo al registro **w**). También puede rotar o desplazar los bits a la izquierda o a la derecha usando **rl** (izq.) y **rr** (derecha). A diferencia de otras instrucciones lógicas, estos comandos operan sobre un solo registro (o el registro **w** en el caso de **not**). Cuando desplaza un registro a la izquierda, cada bit se mueve a la izquierda. Así el bit 7 obtiene el valor del bit 6, el bit 6 obtiene el del bit 5, etc. El bit 0 obtiene el valor del bit de acarreo, y el acarreo obtiene el valor del bit 7. Desplazando a la derecha se invierte el proceso, el bit 7 obtiene el valor del acarreo, y el bit 0 se desplaza al bit de acarreo.

Tip: Cuando desplaza a la izquierda, multiplica el número por 2. Desplazar a la derecha es lo mismo que dividir por 2.

Combinando desplazamientos y sumas puede realizar multiplicaciones. Por ejemplo, suponga que quiere multiplicar un número por 10 (no es tan improbable). Un método sería sumar 10 veces el mismo número en un bucle. Si bien esto funcionaría, un método más eficiente sería observar que multiplicar por 10 es lo mismo que multiplicar el número por 8, multiplicar el mismo valor por 2 y luego sumar ambos productos. Dado que 8 y 2 son potencias de 2, puede hacer estos productos con desplazamientos.

Este es un ejemplo de ambos estilos de multiplicación:

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

                  org         8      ; inicio de datos
valor             ds          1
resultado         ds          1
resultado2        ds          1
contador          ds          1
                  watch valor,8,udec
                  watch resultado,8,udec
                  watch resultado2,8,udec
                  val = 21

                  org         0          ; inicio de código

inicio                             ; multiplica de 2 formas por 10
                  mov         valor,#val        ; producto mediante suma
                  mov         contador,#10
                  clr         resultado
                  mov         w,valor
bucle             add         resultado,w       ; guarda resultado
                  djnz        contador,bucle    ; repite 10 veces
                  nop
                  mov         valor,#val        ; ahora por desplazamiento
                  clc
                  rl          valor             ; valor = valor *2
                  mov         resultado2,valor  ; resultado parcial
                  clc
                  rl          valor
                  clc
                  rl          valor             ; valor = valor *8
                  add         resultado2,valor  ; guarda resultado2

                  sleep
```

Tip: No olvide limpiar el acarreo antes de rotar el registro cuando multiplique o divida. Caso contrario el bit de acarreo ingresará dentro de la palabra que está desplazando, arruinando el resultado.

Por supuesto, si no puede descomponer el producto en algo que pueda realizar con rotaciones o desplazamientos, tendrá que emplear las técnicas que se explicarán en la próxima unidad. Desafortunadamente, no hay una forma simple de realizar cocientes (divisiones). Puede dividir por 2, 4, 8, o cualquier potencia de dos, pero no hay una forma fácil de dividir por 10 o cualquier otro número arbitrario.

Resumen

¡Bueno! Esta unidad abarcó muchos temas. Aprendió sobre **ADD**, **SUB**, **ADDB**, **SUBB**, muchas operaciones sobre bits, e incluso algunos saltos condicionales. Usando estas instrucciones puede hacer muchas cosas diferentes incluyendo matemática simple, controlar la cantidad de veces que se ejecuta un segmento del programa, y comparar números. Estas son las estructuras fundamentales que le permiten al microcontrolador tomar sus propias decisiones.

Recuerde que en la Unidad I leyó que la computadora lee sus entradas, procesa la información, y genera salidas. Las instrucciones de este capítulo son las que usará cuando realice el procesamiento.

Ejercicios

1. Modifique el programa del contador de forma que use **inc** en lugar de **add**. ¿Aún necesita **addb**? En caso afirmativo, ¿qué bit debería sumar?
2. Modifique el contador de forma que realice una cuenta de 32-bit en lugar de dos bytes. Pruebe sus cambios usando el debugger (depurador).
3. Escriba un programa que genere un tono de 1kHz en un parlante conectado al pin 7 del puerto B.

Nota: no conecte un parlante común directamente a los pines de salida del SX. Use un parlante piezoeléctrico diseñado para ser alimentado directamente por CI (Circuitos Integrados). Si es posible, mida la salida con un osciloscopio o un frecuencímetro.

Respuestas

4

1. Su usa **inc**, recuerde que **inc** no afecta el indicador de acarreo. Sin embargo, si lo hace con el de cero. Si incrementa un número y obtiene un cero, es razonable pensar que ocurrió un desborde. El código correcto podría verse así:

```
inc contador
addb contador+1,status.2  ; status.2 es la bandera de cero
```

2. El problema se reduce a modificar la instrucción **ds** para que reserve 4 bytes en lugar de 2 y agregar dos instrucciones **addb** más, a continuación de la preexistente:

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

3. Esta es una posible solución:

```
            device      pic16c55,oscxt5
            device      turbo,stackx_optionx
            reset       inicio
            freq        50000000    ; 50 Mhz
            org         8           ; inicio datos
pausa       ds          1
pausa1      ds          1
            org         0           ; inicio código
inicio
            mov         !rb,#$7F    ; solo el parlante es salida
bucle       not         rb          ; invierte bits
            mov         pausa1,#24
bucle1      clr         pausa
bucle2      djnz        pausa,bucle2
            djnz        pausa1,bucle1
            jmp         bucle
```

Los programas y la información de este texto son para propósitos educativos, habiendo sido cuidadosamente probados, pero no están garantizados para ningún propósito en particular. El editor, el autor y el traductor no ofrecen ninguna garantía sobre la precisión, adecuación , o completitud de la información presentada, y no asumen ninguna responsabilidad por errores u omisiones en el texto. Tampoco asumen la responsabilidad por daños emergentes por el uso de la información de este texto, o por cualquier violación de los derechos intelectuales o de autor de terceros, que resulten del uso de esta información.

Versión en Castellano 1.0

Unidad V. Control de Flujo Avanzado

Unidad V de Introducción al Lenguaje Assembler con el Microcontrolador Scenix SX

© 1999 Parallax, Inc. Reservados todos los derechos. Autor Al Williams, AWC. Traducción Arístides Álvarez.

En la unidad anterior, aprendió a controlar el flujo de la ejecución en función de condiciones. Instrucciones como **jz**, **jc**, y **djnz** le permiten saltar cuando se encuentra cierta condición. Hay otras formas de controlar el flujo de su programa sin embargo, cosa que aprenderá en esta unidad. Además, verá varios métodos para realizar multiplicación y división entera.

Salteando

Todas las instrucciones de salto que aprendió en la unidad anterior en realidad no son instrucciones de código de máquina, sino que son pequeñas subrutinas que el compilador escribe por usted. La única tarea que puede realizar el SX es un control condicional que puede saltear una instrucción. La idea es ejecutar una instrucción que, dependiendo de una condición, ejecutará o salteará la siguiente instrucción. Si esta instrucción fuese un **jmp** entonces obtendría un equivalente de la instrucción de salto que ha usado anteriormente.

Hay dos cosas que considerar en este punto. Primera, la instrucción salteada no tiene que ser necesariamente un **jmp**. Esto puede favorecer la creación de código más eficiente o más rápido en algunos casos. La segunda es que, sin embargo, la función **skip** se saltea una sola instrucción de código de máquina. Muchas de las instrucciones que usa en assembler en realidad consisten de varias instrucciones en código de máquina que son generadas por el compilador (ver Tabla V.I).
Por ejemplo, algunas instrucciones **mov** necesitan dos palabras. Considere este segmento de código:

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

La instrucción **skip** se supone que hace saltar al SX a la instrucción siguiente sin ninguna condición. Sin embargo, solamente hace que se saltee la siguiente instrucción en código de máquina. No hay una instrucción de código de máquina que ejecute **mov** de una constante (o *literal*) a un registro que no sea **w**. Así que el compilador en realidad genera:

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

El resultado es que el programa guarda el contenido de **w** (cualquiera sea) en el registro 8 sin cargar primero 100 en él. Esto no era lo que usted esperaba. Por esta razón, debe ser muy cuidadoso cuando usa **skip**.

Como habrá notado, saltear una instrucción incondicionalmente no tiene mucho sentido. Lo que realmente quiere es una instrucción que saltee según una condición. Hay seis instrucciones **skip** de este tipo. Las instrucciones **sb** y **snb** saltean una instrucción según el valor de un bit. El assembler también provee instrucciones especiales para la bandera de acarreo (**sc** y **snc**), y la de cero (**sz** y **snz**).

Instrucción	Palabras
ADD (sin W)	2
ADDB	2
AND (sin 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 (algunas formas)	2
MOVB	4
OR (sin W)	2
RETW (con valores múltiples)	varía
SUB (sin W)	2
SUBB	2
XOR (sin W)	2

Tabla V.1 – Instrucciones de Varias Palabras

Comparando

Por supuesto, una acción muy común es comparar dos valores y según el resultado saltar a algún punto del programa. Hizo esto en la unidad anterior con una resta y una instrucción de salto. El compilador le permite usar una notación resumida para realizar comparaciones de varias instrucciones en una sola. Puede encontrar una lista de éstas en la Tabla V.2. Estas instrucciones requieren tres piezas de información: un registro, otro registro o una constante, y una dirección a donde saltar.

Instrucción	Uso	Equivalente Basic	Forma de Skip
CJA A,B,LBL	Salta si es mayor	If A>B then LBL	CSA
CJAE A,B,LBL	Salta si es mayor o igual	If A>=B then LBL	CSAE
CJB A,B,LBL	Salta si es menor	If A<B then LBL	CSB
CJBE A,B,LBL	Salta si es menor o igual	If A<=B then LBL	CSBE
CJE A,B,LBL	Salta si es igual	If A=B then LBL	CSE
CJNE A,B,LBL	Salta si es distinto	If A<>B then LBL	CSNE

Tabla V.2 – Instrucciones de Comparación

Estas instrucciones de comparación son muy parecidas al comando **if** de Basic o C. La única diferencia es que la comparación se realiza sobre dos variables o una variable y una constante. Encontrará la sintaxis Basic equivalente en la Tabla V.2.

También puede realizar una comparación y saltear la instrucción siguiente si la comparación es verdadera. Sólo que como con cualquier instrucción skip, no debe intentar saltear una instrucción de más de una palabra (ver Tabla V.1). La Tabla V.2 muestra las instrucciones skip.

Usando Call y Return

A menudo deberá realizar las mismas acciones varias veces dentro de un programa. Por ejemplo, si quiere sumar dos números de 16-bit, seguramente lo volverá a hacer en algún otro punto del programa.

El SX le permite escribir código de forma que pueda ser reutilizado mediante las instrucciones **CALL** y **RET**. Estas instrucciones realizan la misma tarea que **GOSUB** realiza en Basic (o functions en C).

En la unidad anterior, hay un programa que genera un tono de 1kHz en un parlante conectado al pin 7 del puerto B. Ahora suponga que necesite un programa que realice lo siguiente:

1. Emita sonido por el parlante durante 1 segundo
2. Espere a que se presione un botón en el puerto B, pin 0
3. Emita el sonido nuevamente (1 segundo)
4. Regrese al paso 2

Puede encontrar el circuito necesario para este ejemplo en la Figura V.1. El código que generaba el tono de 1kHz se veía así:

```
bucle           not   rb        ; invierte bits
                mov   pausa1,#24
bucle1          clr   pausa
bucle2          djnz  pausa,bucle2
                djnz  pausa1,bucle1
                jmp bucle
```

Debido a que cada bucle requiere aproximadamente 500uS, deberá ejecutar el bucle 2000 veces para generar un tono de 1 segundo. Esto simplemente necesita otro bucle. Sin embargo, es un desperdicio de espacio de programa duplicar este código en dos lugares diferentes del programa. Este es el punto donde la instrucción **call** se vuelve útil. Puede hacer una *subrutina* con el código que genera el tono y luego llamarlo desde diferentes puntos del programa.

Para crear una subrutina, simplemente agréguele un rótulo al código. Otras partes del programa usarán este rótulo (junto con **call**) para ejecutar la subrutina. Cuando la subrutina ejecuta una instrucción **ret** (return = regreso), finaliza la subrutina y continúa con la línea de código posterior a **call**. Este es el código del tono escrito como subrutina:

```
tono            mov   segundo,#$DO        ; 2000 es $7D0
                mov   segundo+1,#$07
bucle           not   rb                  ; invierte bits
                mov   pausa1,#24
bucle1          clr   pausa
bucle2          djnz  pausa,bucle2
                djnz  pausa1,bucle1
                                          ; repite 2000 veces
                djnz  segundo,bucle
                djnz  segundo+1,bucle
                ret                       ; vuelve de donde fue llamado
```

Ahora la parte principal del código simplemente usará **call tono** en cualquier lugar donde se desee un tono de un segundo. Es perfectamente aceptable tener más de un punto de llamada a la subrutina. Por ejemplo, si desea modificar el valor de la variable **segundo** en su programa principal, podría llamar a **bucle** en lugar de a **tono** (aunque probablemente preferiría cambiar el nombre de este rótulo por uno más representativo). Podría generar un tono de la mitad de duración así:

```
                mov   segundo,#$E8        ; 3E8
                mov   segundo+1,#$03
                jmp   bucle
```

Las subrutinas pueden llamar a otras subrutinas, pero el SX solamente puede manejar 8 niveles de subrutinas *anidadas*. Esto quiere decir que, si la subrutina A llama a la B, y la subrutina B llama a la C, y así, el SX se confundirá cuando la subrutina H llame a la I.

Tip: Esto de ninguna manera limita la cantidad de subrutinas que puede tener un programa. Simplemente limita el número de subrutinas que pueden estar activas a la vez.

Para ayudarle a comprender la idea de subrutinas anidadas y el límite en la cantidad, piense en un ascensor que puede llevar 8 personas máximo. Cada vez que ejecuta una instrucción **call**, está poniendo una persona más en

el ascensor. Por cada instrucción **ret** (o una instrucción **retw**; ver más adelante), alguien se baja del ascensor. Si ejecuta 8 instrucciones **call** seguidas, el ascensor se llena y no puede agregar más personas hasta que alguien se baje. Sin embargo, muchas personas pueden haber usado el ascensor (incluso más de una vez). Mientras que no sean más de 8 a la vez, todo funciona.

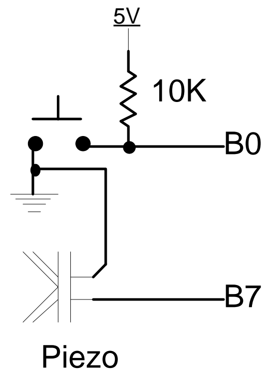

Figura V.1 – Un Parlante Piezoeléctrico y un Pulsador Conectados al SX

Este es el programa completo:

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

            org         8
segundo     ds          2            ; cuenta 1 segundo
pausa       ds          1
pausa1      ds          1

            org         0
inicio      mov         !rb,#$7F     ; parlante es salida
            call        tono

boton       jb          rb.0,boton   ; espera que se presione pulsador
            call        tono
            jmp         boton
```

```
; subrutina
tono        mov         segundo,#$d0        ; 2000 es $7D0
            mov         segundo+1,#$07

bucle       not         rb                  ; invierte bits
            mov         pausa1,#24
bucle1      clr         pausa
bucle2      djnz        pausa,bucle2
            djnz        pausa1,bucle1
            djnz        segundo,bucle       ; repite 2000 veces
            djnz        segundo+1,bucle
            ret                             ; regresa
```

Tip: ¿Qué pasa si quiere usar esta subrutina en un programa que ya tiene los rótulos **bucle1**, **bucle** y **bucle2**? Para evitar conflictos, use rótulos locales (como **:bucle1**, **:bucle** y **:bucle2**) en sus subrutinas.

Es necesario hacer algunas aclaraciones sobre este programa. Lo más notable es que es el primer programa de este libro que lee una entrada. El pulsador está conectado de tal forma que el bit 0 del puerto B leerá un 0 cuando presione el botón. La instrucción **jb** se encarga de esta tarea. Si el bit es un 1, salta a **botón**.

Los pulsadores son dispositivos mecánicos, y por lo tanto tienen *rebote*. Esto significa que cuando presiona el botón, el SX puede ver al pulsador abierto y cerrado muchas veces en unos pocos microsegundos, hasta que queda definitivamente cerrado. Lo mismo sucede cuando suelta el botón. Este parece encenderse y apagarse rápidamente hasta que finalmente queda en la posición apagado. En este programa, esto no es un problema debido a que el tono fuerza una pausa de un segundo antes de que el SX lea el pulsador nuevamente. Sin embargo, si por algún motivo necesita leer rápidamente el estado del botón, deberá tener en cuenta este rebote mecánico.

Si ejecuta este programa y mantiene presionado el botón, el tono continuará sonando hasta que lo libere. Esto es porque el programa no espera a que libere el botón para continuar.

A menudo se desea que la subrutina entregue algún dato (por ejemplo el código de estado) en el registro **w**. Para realizar esta tarea común, el SX provee la instrucción **retw**. Esta carga una constante en el registro **W** antes de regresar. Así:

```
      retw  #$FF
```

es lo mismo que:

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

Por supuesto, **retw**, es solamente una instrucción simple así que se ejecuta más rápido y usa menos espacio.

Tablas

Un uso importante de **retw** es la generación de tablas. Suponga que quiere hallar el cuadrado de un número entre 0 y 10. Ya sabe que la multiplicación es difícil de hacer, así que tiene sentido almacenar los valores en una tabla y leerlos directamente, en vez de realizar los cálculos. Esta es una subrutina que hace eso:

```
; cuadrado de un número de 0 a 10 en el registro W
; regresa el resultado en el registro W
cuadrado     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
```

Cuando el programa llama a la subrutina **cuadrado**, salta a la instrucción **retw** correspondiente al valor de **W**. La instrucción **retw** carga el valor correcto en **W** y regresa al programa principal. Esto es simple, eficiente, y muy rápido. También es muy común que el assembler le permita escribir múltiples valores en la misma línea. Así que podría reemplazar la subrutina **cuadrado** con dos líneas de assembler:

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

El código de máquina generado es el mismo en ambos casos, así que no hay diferencia en usar cualquier método. Es una cuestión de gustos personales.

Apuntador o índice

Cuando accede a los registros del SX, necesita conocer de antemano la dirección en la que se encuentran. Anteriormente en este libro, usó direcciones numéricas (como 8 o 9), pero rápidamente vio la ventaja de usar nombres simbólicos (como **status** o **counter**). Sin embargo, algunas veces no conoce la dirección exacta que debe usar. Por ejemplo, suponga que desea limpiar toda la memoria del usuario en el SX. Podría escribir:

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

Sin embargo, eso no parece muy eficiente. Sería bueno poder usar un bucle para *apuntar* a los diferentes registros. Ese es el propósito de los registros especiales **FSR** (File Select Register=Registro de Selección de Archivo) e **IND** (Apuntador ó Índice). El registro **IND** no es un registro común. En realidad es un sobrenombre o alias para algún registro del SX. ¿Cuál? Aquel cuya dirección figura en el **FSR**.

Este es un ejemplo simple:

```
R1    EQU   10    ; registro 10 es R1
R2    EQU   11    ; registro 11 es R2
      mov   R1,#100
      mov   R2,#200
      mov   FSR,#10  ; almacena la dirección 10 en FSR
      mov   w,IND
; W ahora contiene 100
      inc   FSR    ; va a la siguiente dirección
      mov   w,IND
; W ahora contiene 200
      mov   FSR,#R1
      mov   w,IND
; W contiene 100 otra vez
      clr   IND   ; R1 ahora es cero
```

Observe que puede leer o escribir en **IND**. **IND** es el nombre del registro cuya dirección está almacenada en **FSR**, cualquiera sea esta.

Tip: Siempre es preferible cargar un número constante en **FSR**. En el programa anterior, por ejemplo, si hubiese usado: **mov FSR,R1** hubiera cargado el contenido de **R1** (100) en **FSR** y no lo que se pretendía (10).

Este programa limpiará todos los registros del usuario en un bucle:

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

Se toma ventaja del hecho de que cuando **FSR** llega a $20 (esto es, el bit 5 vale 1 por primera vez) el bucle se detiene. También se podría haber comparado **FSR** con $20 o usar cualquier otro esquema para detener el bucle.

Esta técnica no es solamente para limpiar memoria. Cuando programa, a menudo necesitará manejar una lista de datos (por ejemplo, los últimos 4 valores obtenidos de un sensor, o los últimos 8 bytes leídos de un puerto serial). Usar apuntadores es la forma más eficiente de procesar pilas, listas, y otras estructuras de datos.

Funciones Matemáticas

Armado con la habilidad de realizar bucles y controlar valores, puede resolver con facilidad problemas de multiplicación y división arbitrarios. La aproximación más simple para multiplicar, por ejemplo, 9 por 7 es sumar acumulativamente 9 veces 7. Sin embargo, conociendo un poco sobre números binarios, puede escribir un algoritmo más inteligente.

¿Recuerda cómo aprendió a multiplicar en la escuela? Escribía su problema y multiplicaba dígito por dígito, moviendo un lugar a la izquierda en cada paso. Luego sumaba todos los resultados parciales para obtener la respuesta correcta. La computadora también puede hacer esto. Como ventaja extra, el SX trabaja en binario así que cada resultado parcial es cero o el mismo número original desplazado a la izquierda cierta cantidad de veces. Piense cómo multiplicar %1001 por %101 (9 por 5).

```
          1001
X          101
----------------
          1001
         0000
+       1001
----------------
        101101 = 32 + 8 + 4 + 1 = 45
```

Realizar la multiplicación de esta forma se conoce como algoritmo de Booth (un *algoritmo* es el nombre que se le da a una secuencia de pasos de programa). Este es un segmento del código del SX que multiplicará el byte del registro **V1** por el byte del registro **V2**:

```
      clr V3            ; puesta a cero del resultado
      mov ctr,#8        ; 8 bits
```

```
bucle       rr V2              ; carga el bit 0 de V2 en el bit de acarreo
            jnc nosuma         ; saltea la suma si acarreo = 0
            add V3,V1          ; sumar al resultado
nosuma      rl V1              ; desplaza V1 un lugar
            djnz ctr,bucle     ; repite 8 veces
```

Por supuesto, el resultado (**V3**) es de un byte, así que no podrá multiplicar números que requieran una respuesta mayor que 255. Puede extender fácilmente este algoritmo para manejar más bits.

División

Puede usar un algoritmo similar para la división. Si recuerda la nomenclatura de la escuela, en la división participan el *divisor*, el *dividendo*, dando como resultado el *cociente*. Así cuando calcula 20 dividido por 5, 20 es el dividendo y 5 el divisor. El resultado, 4, es el cociente. Debido a que 5 entra en 20 una cantidad entera de veces (4), el *resto* es 0.

Cuando realiza una división en un papel, la reduce a una serie de restas. También debe ir desplazando su posición para ir examinando los dígitos. El SX puede hacer lo mismo en binario. Dado que en binario sólo hay unos y ceros, es fácil decir si un número "entrará" en otro; simplemente controle si el primer número es menor o igual que el segundo.

Considere estos pasos de programa (o algoritmo, si lo prefiere):

1) Ponga el cociente en 0
2) Desplace el divisor a la izquierda hasta que el bit superior sea un 1
3) Recuerde cuántos desplazamientos realizó en el paso 2 y agregue 1 a esa cuenta
4) Desplace el cociente a la izquierda (multiplica por 2)
5) Compare el dividendo con el divisor; si el dividendo es mayor o igual que el divisor, reste el divisor del dividendo y sume 1 al cociente
6) Desplace el divisor a la derecha
7) Reste 1 del contador y, si no es cero, regrese al paso 4

Suponga que quiere dividir 20 por 5. Después de realizar los pasos 1 a 3, tendrá el divisor en 160 y el contador en 6. Esta es la parte del bucle del algoritmo después de realizar el paso 6:

Dividendo	Divisor	Cociente	Contador	Comentarios
20	160	0	6	Desplazó 5 ceros; No resta
20	80	0	5	No resta
20	40	0	4	
0	20	1	3	Resta
0	10	2	2	
0	5	4	1	

¿Qué sucede con una división con resto? Si reemplaza el 20 de la tabla anterior por, por ejemplo, 22 verá que la columna del dividendo tendrá un 2 luego de la resta. Dado que el divisor nunca es menor que 2, la respuesta es la misma. Sin embargo, la columna del dividendo queda con el valor del resto (2).

Este es un programa de división simple escrito para el SX:

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

            org         8
dividendo   ds          1
divisor     ds          1
cociente    ds          1
contador    ds          1
            watch dividendo,8,udec
            watch divisor,8,udec
            watch cociente,8,udec
            watch contador,8,udec

            org         0
inicio
            mov         dividendo,#20
            mov         divisor,#5
            call        divide
            break
            nop
            sleep

; subrutina

divide      clr         contador    ; asume que no divide por cero
            clc
:bucle      rl          divisor
            inc         contador
            jnc         :bucle
; deja al divisor con el bit superior en 1
            rr          divisor
; contador tiene el número de bits
            clr         cociente
:bucle1
```

```
            test        contador
            jz          :listo
            clc
            rl          cociente
            cjb         dividendo,divisor,:bucle2
            sub         dividendo,divisor
            inc         cociente
:bucle2
            dec         contador
            clc
            rr          divisor
            jmp         :bucle1
:listo
            ret         ; regresa al programa principal
```

Una cosa que este programa no hace es evitar la división por cero, que es un error. Es muy simple agregar una instrucción **test** para controlar si **divisor** es cero y saltar a una subrutina de error.

Resumen

En esta unidad aprendió sobre instrucciones que comparan dos valores y toman una decisión en base al resultado de ésta. Este tipo de control de flujo es crucial para desarrollar algoritmos avanzados de multiplicación y división (así como también para muchas otras tareas de programación). Esta unidad también presentó las subrutinas (mediante las instrucciones **call** y **ret**) y la forma de emplearlas para realizar tablas de constantes. También puede crear tablas usando los registros del apuntador o índice (**fsr** e **ind**) que le permiten acceder a los registros mediante un código más sencillo.

En este punto del libro, tiene las herramientas necesarias para escribir algunos programas poderosos. En las próximas tres unidades aprenderá cómo emplear toda la memoria del SX y cómo controlar el hardware. Además, trabajará con interrupciones y periféricos virtuales.

Ejercicios

1. El programa de ejemplo de esta unidad emite un tono cuando el botón es presionado brevemente. Sin embargo, si el botón permanece presionado, el tono continúa. Modifique el programa de forma que después de emitir un tono, espere hasta que se libere el botón. Asegúrese de incluir los pasos necesarios para eliminar el rebote del pulsador.
2. Cuente la cantidad de veces que se presiona el botón. Después de 10 veces, ponga el controlador a dormir (sleep).
3. En unidades anteriores, hay un programa que hace titilar leds usando **sleep** y el temporizador watchdog para obtener la pausa entre cada parpadeo. Sin embargo, esto no permite inicializar los LEDs en un estado conocido debido a que el programa no puede ver la diferencia entre el primer reset y un reset después de una instrucción **sleep**. Observe que el bit 4 del registro **status** es 0 cuando actúa el

watchdog. Cambie el programa para inicializar el puerto B en $AA en el caso de un reinicio. El programa original se muestra a continuación.

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

            org         8
patron      ds          1

            org         0

inicio      mov         !rb,#0    ; todo el puerto b salidas
            xor         patron,#$FF
            mov         rb,patron
            sleep
```

4. Conecte botones (como se muestra en la Figura V.1) a los pines 0, 1, 2 y 3 del puerto B. Conecte un parlante piezoeléctrico al pin 7 del puerto B. Realice un programa que toque un tono diferente durante 500mS cada vez que presiona un botón. Con más botones, ésta sería la base de un timbre musical o de un piano de juguete para niños.

Respuestas

1. Este es el código principal:

```
inicio          mov          !rb,#$7F    ; parlante es salida
                call         tono
   ; espera que se presione el botón
boton           jb           rb.0,boton
                call         tono
soltar          jnb          rb.0,soltar
; espera que se termine el rebote
                clr          pausa
:rebote         djnz         pausa,:rebote
                jmp          boton
```

La pausa le da tiempo al pulsador para que deje de causar rebote. El tiempo es arbitrario y puede necesitar ajuste dependiendo del tipo de interruptor que use.

5

2. Este es un extracto de la solución:

```
         org   8
segundo        ds          2    ; contador para 1 segundo
pausa          ds          1
pausa1         ds          1
presionado     ds          1


               org         0
inicio         mov         !rb,#$7F    ; parlante es salida
               call        tono
               clr         presionado
                                 ; esperar que se presione el botón
boton          jb          rb.0,boton
               call        tono
               inc         presionado
               cje         presionado,#10,parar
soltar         jnb         rb.0,soltar
; espera que finalice el rebote
               clr         pausa
:rebote        djnz        pausa,:rebote
               jmp         boton

parar          sleep
```

Por supuesto, también sería correcto almacenar 10 en la variable presionado y disminuir su valor hasta cero. Esto sería más eficiente debido a que revisaría la bandera de cero después de restarle 1 a la variable, ahorrando así un paso.

3. La solución es simplemente verificar si el bit 4 es 0:

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

               org         8
patron         ds          1

               org         0
```

```
inicio   mov         !rb,#0    ; todo el puerto b salidas
; controla si es reinicio o inicia por primera vez
         jnb         status.4,agn
         mov         patron,#$AA
agn      xor         patron,#$FF
         mov         rb,patron
         sleep
```

Podría argumentar que es conveniente asignar a **patron** $55 en lugar de $AA debido a que la instrucción siguiente invertirá los bits, pero de cualquier forma el resultado es aceptable.

4. Hay varias formas de resolver este ejercicio, dependiendo de preferencias personales. Lo que hay que tener en cuenta es que debido a que cada tono tiene un período diferente, debe ajustar la cantidad de ciclos para obtener 500mS. Por ejemplo, un tono de 1kHz tiene ciclos de 500uS, así que se necesitan 1000 ciclos para obtener 500mS. Sin embargo, un tono de 2kHz tiene ciclos de 250uS y por lo tanto requiere 2000 ciclos para mantener la misma duración. Esta es una solución:

```
            device      pic16c55,oscxt5
            device      turbo,stackx_optionx
            reset inicio
            freq        50000000   ; 50 Mhz
            org         8
segundo     ds          2   ; contador para 1 segundo
pausa       ds          1
pausa1      ds          1
tono        ds          1   ; constante de tono
            org         0
inicio      mov         !rb,#$7F    ; parlante salida
      ; espera a que se presione el botón
boton       jnb         rb.0,bp0
            jnb         rb.1,bp1
            jnb         rb.2,bp2
            jb          rb.3,boton
; tono 3
            mov         tono,#48
            mov         segundo,#$01
            mov         segundo+1,#$01

bp          call        sonido
            jmp         boton

bp2         mov         tono,#24
```

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

bp1         mov         tono,#12
            mov         segundo,#$FA
            mov         segundo+1,#$03
            jmp         bp

bp0         mov         tono,#6
            mov         segundo,#$F4
            mov         segundo+1,#$07
            jmp bp

; subrutina
sonido
bucle       not         rb                 ; invierte bits
            mov         pausa1,tono
bucle1      clr         pausa
bucle2      djnz        pausa,bucle2
            djnz        pausa1,bucle1
            djnz        segundo,bucle
            djnz        segundo+1,bucle
            ret                      ; regresar
```

Los programas y la información de este texto son para propósitos educativos, habiendo sido cuidadosamente probados, pero no están garantizados para ningún propósito en particular. El editor, el autor y el traductor no ofrecen ninguna garantía sobre la precisión, adecuación, o completitud de la información presentada, y no asumen ninguna responsabilidad por errores u omisiones en el texto. Tampoco asumen la responsabilidad por daños emergentes por el uso de la información de este texto, o por cualquier violación de los derechos intelectuales o de autor de terceros, que resulten del uso de esta información.

Versión en Castellano 1.0

Unidad VI. Programación de Bajo Nivel

Unidad VI de Introducción al Lenguaje Assembler con el Microcontrolador Scenix SX

© 1999 Parallax, Inc. Reservados todos los derechos. Autor Al Williams, AWC. Traducción Arístides Álvarez.

En las unidades anteriores ha escrito programas simples que emitían salidas o leían entradas. Sin embargo, el SX tiene muchas características poderosas de E/S que usted puede usar, si sabe como trabajan. Además de las capacidades de entrada y salida, el SX tiene mucho más espacio para almacenar programas y datos que el que ha usado hasta ahora. Para acceder a esta memoria extra, necesitará comprender una técnica especial llamada *banking*.

Control de Puertos

El SX tiene tres puertos de E/S: puertos A, B, y C. Por supuesto, el dispositivo de 18-pines no tiene puerto C, pero en el resto de sus características es igual a sus primos de encapsulados más grandes. El puerto A tiene solamente 4 pines. Los puertos B y C tienen 8 bits cada uno. Puede leer o escribir los pines de un puerto, accediendo al registro de datos correspondiente (**ra**, **rb**, o **rc**). También ha visto que puede cambiar la dirección de cada pin modificando el registro de control del puerto (**!ra**, **!rb**, o **!rc**).

Sin embargo, el registro de control le da mucho control, más que simplemente la dirección, sobre los puertos. Usando el registro de control puede fijar otras opciones incluyendo el umbral de tensión para cada pin, o determinar si el pin usa una entrada de Schmitt trigger o una entrada de nivel lógico normal. También puede elegir conectar un resistor de pull-up opcional en cada pin.

¿Cómo puede un simple registro de control tener tantas opciones? No puede. El truco es que el registro de control tiene múltiples personalidades determinadas por el registro de modo o **M**. Por defecto, el registro **M** (un registro de 4-bits) contiene el valor $F, lo que convierte al registro de control en un registro de dirección. Cuando escribe un 0 en el registro de control, hace al bit correspondiente una salida, y un 1 hace una entrada.

Si pone en el registro de modo $E, por ejemplo, el registro de control selecciona qué pines tienen un resistor de pull up conectado internamente. Cada bit que tiene cero conectará el resistor de pull up. Estos resistores evitan que los pines de entrada asuman valores aleatorios si no tienen un circuito externo que los alimente. Puede fijar estos resistores en cualquiera de los tres puertos, poniendo en **M** el valor $E y luego modificando los registros **!ra**, **!rb**, o **!rc**.

Puede usar la instrucción **mov** para cargar al registro **M** con el contenido de otro registro o un valor literal. También puede usa la instrucción **mode** para cargar un valor en **M**. La Tabla VI.1 muestra el efecto del registro de control para distintos valores de **M** (observe que esta tabla no muestra las configuraciones que corresponden a las interrupciones, tema que se tratará en la próxima unidad).

Modo	!ra	!rb	!rc	Nombre SX
$F	Dirección	Dirección	Dirección	TRIS_
$E	Pull up	Pull up	Pull up	PLP_
$D	Umbral	Umbral	Umbral	LVL_
$C	N/A	Schmitt	Schmitt	ST_

Tabla VI.1 – Configuraciones de Modo (mode)

Si pone el bit de umbral en 0, el SX leerá las entradas a través de un buffer compatible con CMOS. Este buffer tratará a los niveles inferiores al 30% de la tensión de fuente (digamos 1.5V si la fuente es 5V) como un 0. Todo lo que esté por encima del 70% de la tensión de fuente (3.5V) será un 1. Las tensiones intermedias generarán valores impredecibles, aunque la experiencia práctica muestra que el umbral está aproximadamente al 50% de la tensión de la fuente (pero Scenix no especifica esto).

Cuando el bit de umbral es 1, la entrada usa un buffer compatible con TTL. Usando esta configuración se toma como 0 a 0,8V o menos y como un 1 cualquier valor por encima de 2V. Para la mayoría de los circuitos de lógica moderna, esto es aceptable, pero la conexión con ciertos dispositivos puede requerir una configuración o la otra. También, cuando mezcla circuitería analógica con el procesador, podría querer ajustar los umbrales para leer un nivel de tensión particular.

Los puertos B y C pueden usar una configuración de entrada Schmitt trigger si pone un cero en el registro Schmitt. Un Schmitt trigger usa diferentes umbrales de acuerdo a la situación. Imagine que intenta regular la temperatura de su pileta a una temperatura en particular (digamos 25 grados Celsius). Enciende el calentador del agua y observa el termómetro, cuando la temperatura llega a 25, lo apaga. Sin embargo, la pileta pierde calor rápidamente, así la temperatura cae casi inmediatamente y se debe encender nuevamente el calentador. De esta forma usted se encuentra encendiendo y apagando el calentador cada pocos segundos, no siendo capaz de mantener los 25 grados por más de una fracción de segundo.

Un Schmitt trigger usa *histéresis* para resolver este problema. La idea es que el Schmitt trigger usará un umbral para identificar las transiciones de 0 a 1 y otro para las transiciones de 1 a 0. Un Schmitt trigger podría ver una tensión que aumenta de 0,8 a 0,9V y emitir un 1 lógico (5V). Sin embargo, se necesitaría que la tensión cayera por debajo de 0,5V para regresar al estado cero (0V). Esto evita la creación de señales ruidosas o inestables que causarían múltiples cambios en el estado de la salida. El Schmitt trigger del SX usa el 15% y el 85% de la tensión de fuente como puntos de cambio. Una vez que la señal sobrepasa el 85% de la tensión de fuente (4,25V para una fuente de 5V), la entrada lee un 1. Seguirá leyendo un 1 hasta que la entrada caiga por debajo del 15% (0,75V).

Esto puede ser importante cuando trabaja con señales de sensores del "mundo real", o con entradas ruidosas provenientes de líneas largas. También puede usarse para “rectificar” una señal, por ejemplo, leyendo una entrada digital de un capacitor cargándose. Por supuesto, al usar la opción Schmitt trigger se modifican las configuraciones de umbral de los pines.

Tip: Asegúrese de conocer el estado del registro **M** antes de usar los registros de control. Un error común es fijar un valor del registro **M** distinto a $F, usar el registro de control, y después en otro punto del programa intentar usar el registro de control para fijar la dirección de los bits. El registro **M** queda en el último valor que le fue asignado hasta que ocurra un reinicio (reset).

Capacidades Analógicas

El SX tiene una capacidad especial más en el puerto B. Los pines 1 y 2 del puerto B pueden funcionar como un comparador analógico. Puede leer la salida del comparador por software o también puede hacer que el pin 0 del puerto B refleje la salida del comparador.

Para habilitar el comparador, simplemente cargue en el registro **M** un 8 y escriba un valor en **!rb**. Un valor de $C0 apagará la función de comparación. Para encenderla, escriba $40 o $00 en **!rb**. Si usa $00, el comparador funcionará, y el pin 0 actuará como salida del mismo. Si usa $40, el pin 0 estará libre para E/S normal, pero el comparador seguirá funcionando.

6

Para leer el estado del comparador, asegúrese que **M** contiene un 8 y escriba el registro **!rb**. Cuando abre el registro del comparador (esto es cuando **M** es igual a 8 y realiza un **mov** a **!rb**) el SX hace un pequeño truco a sus espaldas. En lugar de simplemente mover los datos al registro del comparador, en realidad intercambia el registro **W** con el registro del comparador. Esto se cumple aunque escriba:

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

Que es lo mismo que escribir:

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

Así que después de escribir en el registro del comparador, el registro **W** contiene el valor anterior. Solamente debería examinar el bit 0, el bit de estado del comparador, luego de haber habilitado el comparador con otra instrucción. Si el bit 0 está en estado alto, entonces la tensión en B2 es mayor que en B1. Si es menor, hay un cero en el bit 0.

¿Para qué puede necesitar un comparador? Tal vez quiera que el SX compare las tensiones de un potenciómetro y una termocupla. Quizás quiera dividir la tensión de la batería y compararla a una referencia conocida para detectar cuando las baterías estén bajas.

Banco de Registros

Anteriormente leyó que el SX tiene más de 100 registros. Eso puede parecer raro, debido a que el debugger muestra solamente 32 registros. Si examina el juego de instrucciones del SX, verá que solamente hay lugar para 5 bits para especificar la dirección de un registro. ¿Cómo pueden 5 bits direccionar más de 100 registros? La respuesta es mediante banking (bancos de memoria).

Su programa tiene acceso a 136 ubicaciones de memoria (sin incluir registros especiales como **ind**, **fsr**, **ra**, etc.). Sin embargo, solo puede trabajar con 32 a la vez. Los primeros 8 (registros 0 a 7) son los registros especiales y siempre pueden ser accedidos. Los registros de 8 a 15 también son accesibles siempre. El SX no los usa para nada, así que los puede emplear para lo que quiera. Esto completa la distribución de los primeros 16 registros. Los otros 16 (registros $10-$1F) están disponibles para cualquier uso. Sin embargo, en realidad hay 8 juegos de estos registros. Los 16 que se están usando los selecciona el registro **FSR**.

Tip: No olvide que en un SX de 18-pines, el registro 7 está disponible para su uso y siempre está disponible. En los otros dispositivos SX, este es el registro **rc**.

Conceptualmente, el mapa de memoria del SX consiste de 8 páginas de 32-bits. Cada página tiene 32 registros. Los primeros 16 siempre son los mismos. Los últimos 16 no. Cada registro tiene su propia dirección (y en el caso de registros compartidos, 8 direcciones). Puede ver esto gráficamente en la Tabla VI.2.

Cuando quiere acceder a un registro, tiene varias opciones. Primera, si está usando **FSR**, simplemente ponga la dirección específica en **FSR** antes de usar **IND**. Así si quiere acceder a la última posición de memoria, cargue el **FSR** con $FF. La segunda opción es cargar los 3 bits superiores de **FSR** antes de acceder a la memoria. Los valores que debe usar están en la fila del encabezado de la Tabla VI.2. Puede almacenar un valor en **FSR**, por supuesto, con una instrucción **mov**. Sin embargo, esto destruye completamente el valor del registro y además requiere dos instrucciones de lenguaje de máquina si está usando un valor literal. Debido a que muchos programas querrán cargar valores literales en **FSR**, hay una instrucción **bank**. Esta instrucción carga los 3 bits superiores de un número en los 3 bits superiores de **FSR**. Esto es útil debido a que puede usar directamente el nombre de la variable a la que quiere acceder. Por ejemplo:

```
            org   $FF
ultima      ds    1

            org   0
            bank  ultima
            mov   ultima,#0
```

Usted se preguntará por qué el debugger no mostró estas páginas extra. En la directiva **device** de todos los programas anteriores, encontrará la cláusula **pic16c55**. Esto le dice al SX que solamente use 1 banco para simular un dispositivo diferente. Si especifica **sx28l** (o **sx18l** para un dispositivo de 18-pines) obtendrá acceso a todos los registros y memoria. El banco actual de registros se muestra en forma resaltada en comparación con los bancos inaccesibles en la pantalla de depuración.

	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 registros direccionables como $08-$0F, $38-$3F, $58-$5F, $78-$7F, $98-$9F, $B8-$BF, $D8-$DF, o $F8-$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

Tabla VI.2 – Mapa de Memoria del SX

Tip: Si organiza los registros de acuerdo a su uso, puede poner nombres significativos a sus bancos. Esto no sólo hace mas legible el código, sino que también reduce la cantidad de movimientos entre bancos. Por ejemplo, suponga que tiene un banco de variables (banco $20) reservadas para cálculos matemáticos, y otro para comunicaciones externas (banco $40). Puede definir dos símbolos, **matemat** y **comunic**, pudiendo usar:

```
bank matemat  ; cambia al banco de variables matemáticas
```

Páginas de Programa

El SX también posee memoria extra para el almacenamiento del programa. Aunque ninguno de sus programas necesitó de ella, el SX tiene 4 páginas de memoria de programa, y cada página tiene una capacidad de 512 instrucciones (recuerde, las instrucciones del SX no son bytes). Así puede usar hasta 2K de instrucciones.

Sin embargo, usar más de 512 instrucciones requiere un planeamiento cuidadoso. Cada instrucción de salto (excepto **jmp w**, **jmp pc+w**, y **ljmp**) solamente usa 9 bits para una dirección. Los bits extra necesarios para el direccionamiento, se obtienen de los 3 bits superiores del registro **status**. En lugar de ajustar manualmente estos bits, sin embargo, puede lograr que el assembler lo haga por usted. Simplemente ponga el caracter "@" delante de la dirección, escribiendo:

```
JMP   @LugarMuyLejano
```

Esto en realidad genera las siguientes instrucciones:

```
PAGE  LugarMuyLejano
JMP   LugarMuyLejano
```

La instrucción **page** configura los bits del registro status para alcanzar esta dirección. Debido a que el símbolo @ requiere espacio extra, solamente debería usarse en los casos en que la dirección de destino esté en una página diferente.

Para complicar las cosas, llamar a una subrutina de otra página es aún más complicado. La instrucción **call** toma una dirección de solamente 8 bits. El noveno bit es puesto en 0, y el resto de los bits se obtienen del registro **status** al igual que con **jmp**. Esto significa que la llamada a una subrutina solamente puede producirse en las primeras 256 instrucciones de una página.

Esto aparenta ser una gran restricción, pero en realidad, es fácil de superar. Si no puede organizar sus subrutinas de forma que entren en la primera mitad de la página, simplemente use un **jmp** a la subrutina (una instrucción simple) en la primera mitad de la página, poniendo la llamada a la subrutina en la primera mitad. No olvide que las tablas de datos (como las que usó en la Unidad 5) en realidad son subrutinas, así que tienen las mismas limitaciones. La instrucción **jmp** que inicia la tabla debe estar en la primera mitad de la página para que otras partes del programa puedan llamar (**call)** a la tabla.

No se dijo nada de que el contador de programa tiene una longitud de 11 bits, pero el registro **pc** son los últimos 8 bits. No hay una forma directa de leer los 3 bits superiores. El único acceso a estos bits se obtiene cuando son cargados desde los 2 o 3 bits superiores del registro **status**.

Cuando llama a una subrutina de una página diferente, necesita que el controlador emplee la dirección completa de 11-bits en el contador de programa. También es útil dejar almacenado el valor del registro **status** para poder realizar más llamadas a subrutinas en la misma página. Este es el propósito de la instrucción **retp**. No sólo restaura la dirección completa para que se retome la ejecución del programa, sino que también almacena los 3 bits superiores de la dirección de retorno en los 3 bits superiores del registro **status**.

Tip: Las instrucciones **ret** y **retp** usan la misma cantidad de espacio y se ejecutan a la misma velocidad. Si hay alguna posibilidad de que una subrutina sea llamada desde otra página, use **retp**. La única excepción sería si se pretende que la subrutina modifique los bits superiores de **status**.

Leyendo un Programa Almacenado

En la unidad anterior vio cómo usar **retw** para hacer tablas en la memoria de programa. Hay otra forma para acceder a la memoria de programa y es mediante la instrucción **iread**. Esta instrucción toma 4 ciclos (inusual para una instrucción que no salta o saltea). Toma los registros **M** y **W** como direcciones de 11-bits, lee la palabra de 12-bits en esa dirección, y la carga en los registros **M** y **W**.

¿Cómo obtener datos arbitrarios de la memoria de programa? Use **DW** como en:

```
            org         0
inicio      mov m,#AlgunDato>>8  ; parte alta de la dirección
            mov w,#AlgunDato&$FF ; parte baja de la dirección
            iread
            nop
            nop
            break
            nop
            sleep


AlgunDato   dw $1A5
```

Si depura este programa verá que el registro **W** tendrá $A5 y el registro **M** tendrá $1 en el breakpoint.

Tip: Sea cuidadoso al acceder a los registros de control de puertos después de ejecutar **iread** dado que el registro **M** no tendrá lo que usted espera y eso altera la función del registro de control.

Resumen

Las técnicas de esta unidad no son tan útiles para programas simples como las que aprendió hasta este punto. Pero en la vida real, con 24 bytes de almacenamiento de datos y 512 instrucciones no se puede llegar tan lejos. La clave para tener éxito con programas grandes es la organización y planificación cuidadosas. Si puede mantener las variables relacionadas en el mismo banco, será más eficiente. Las variables que usa en muchas partes del programa deberían estar antes de $10 (la zona compartida). Por supuesto, con sólo 8 bytes compartidos entre los bancos (9 en un SX18), debe ser muy ahorrativo.

La organización del código también es importante. Rutinas relacionadas en la misma página no necesitan saltos largos. También debe ser cuidadoso para no poner subrutinas en la segunda mitad de cualquier banco, debido a que no podrá llamarlas, al menos directamente.

Si le parece raro que el SX tenga estas formas extrañas de acceso a la memoria, recuerde que esto se hizo en nombre de la compatibilidad. El SX es compatible con controladores anteriores que no tenían tanta memoria. El precio de lograr recursos extra es obtener una complejidad extra.

Ejercicios

1. Escriba un programa para limpiar los 8 bancos de registros. Sea cuidadoso de no limpiar los primeros 8 registros (que son los registros de funciones especiales como **pc** e **ind**). Además, no limpie el banco compartido más de una vez. ¿Puede hacer que el bucle de limpieza sea una subrutina?
2. Use **org $200** para colocar la subrutina de limpieza del programa anterior en el primer banco de programa. Utilice el comando step para ver la ejecución en el depurador.
3. Escriba un programa que convierta la temperatura de grados Celsius a Fahrenheit, usando una tabla que se acceda con **iread**. Asuma un rango de entrada de 0 a 29 grados. La fórmula para la conversión, por las dudas, es F=1,8C+32.

Respuestas

1. Esta es una posible solución:

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

             org         0
inicio       mov         fsr,#8      ; banco compartido
             call        limpiar
             mov         fsr,#$10
bucle1       call        limpiar
             add         fsr,#$11
             jnc         bucle1
             nop
             break
             nop
             sleep

; subrutina que limpia FSR hasta que FSR AND $F es 0
limpiar      clr         ind
             inc         fsr
             mov         w,#$F
             and         w,fsr
             jnz         limpiar
             dec         fsr
             ret                      ; regresa
```

2. Mover la subrutina requiere que: 1) ponga **org $200** delante de la rutina **limpiar**; 2) cambie cada llamada a **limpiar** por **@clear**; y 3) cambie la instrucción **ret** por **retp**. Intente realizar las modificaciones anteriores de a una a la vez y ejecute el depurador para ver cómo se comporta el código antes de realizar el cambio siguiente.

3. Esta es una implementación simple:

```
            org        8
tempm       ds         1   ; lugar temporal para almacenar M
valor       ds         1   ; valor a convertir


            org        0
inicio      mov        valor,#11     ; 11 grados C
            call       @convertir
            nop
            break
            nop
            sleep
convertir   mov        tempm,m
            mov        m,#tabla>>8
            mov        w,#tabla & $FF
            add        w,valor
            iread
            ; no necesita M
            mov        valor,w
            mov        m,tempm       ; recupera M
            ret

tabla       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
```

Los programas y la información de este texto son para propósitos educativos, habiendo sido cuidadosamente probados, pero no están garantizados para ningún propósito en particular. El editor, el autor y el traductor no ofrecen ninguna garantía sobre la precisión, adecuación, o completitud de la información presentada, y no asumen ninguna responsabilidad por errores u omisiones en el texto. Tampoco asumen la responsabilidad por daños emergentes por el uso de la información de este texto, o por cualquier violación de los derechos intelectuales o de autor de terceros, que resulten del uso de esta información.

Versión en Castellano 1.0

Unidad VII. Interrupciones

Unidad VII de Introducción al Lenguaje Assembler con el Microcontrolador Scenix SX

© 1999 Parallax, Inc. Reservados todos los derechos. Autor Al Williams, AWC. Traducción Arístides Álvarez.

Una de las grandes ventajas de las computadoras modernas, es que pueden hacer más de una cosa a la vez. Con una PC bajo Windows, puede navegar por Internet, trabajar en un e-mail, y cargar una foto desde su cámara digital, todo a la vez. Esto suena genial excepto por una cosa: la mayoría de las computadoras (incluida su PC) sólo pueden realizar una tarea a la vez.

¿Cómo es esto posible? Mientras es cierto que la mayoría de las computadoras pueden realizar solamente una tarea a la vez, también es cierto que pueden hacerla muy rápidamente. Los sistemas operativos modernos emplean pequeñas fracciones de tiempo para cada tarea activa. De esta forma, todas las tareas parecen ejecutarse a la vez. Además, las computadoras modernas pueden responder a eventos externos, por ejemplo, un movimiento del mouse o cuando se presiona una tecla. Esto también ayuda a crear la ilusión de que la computadora está realizando varias tareas, debido a que la computadora puede manejar los eventos a medida que ocurren sin provocar demoras.

Para obtener esta capacidad, una computadora necesita una forma de medir el tiempo y también una forma de detener la ejecución de su programa para atender otra tarea. El SX tiene dos características que trabajan juntas en esta área: el *contador de reloj de tiempo real* (registro **RTCC**) e *interrupciones*. El registro **RTCC** hace lo que su nombre implica: se incrementa en intervalos precisos predeterminados independientemente de lo que el controlador esté haciendo. También puede incrementarse en respuesta a una entrada de un pulso externo. Las interrupciones le permiten a un evento externo o a un período de tiempo cumplido, disparar la ejecución de un segmento de programa. Lo que estaba haciendo el SX antes del evento es detenido hasta que se completa la ejecución del código de evento (una *rutina de atención de interrupción* o *isr*).

En la programación en lenguaje assembler, las interrupciones tienen la reputación de ser difíciles de usar. Es cierto que éstas requieren una planificación cuidadosa. Sin embargo, el SX tiene varias características que hacen que el trabajo con interrupciones sea menos problemático en comparación con otros controladores similares.

¿Qué es un evento? Un evento común se produce cuando se desborda el registro **RTCC** (es decir, cambia de $FF a $00). También puede configurar las interrupciones para que se produzcan en los flancos ascendentes o descendentes de cualquiera (o todos) los pines del puerto B. Para usar interrupciones, primero debe configurarlas (por defecto están deshabilitadas).

El Contador de Reloj de Tiempo Real

Una de las fuentes más comunes de interrupción es cuando el valor del registro **RTCC** cambia de $FF a $00. Esto indica que transcurrieron 256 períodos de tiempo o que ocurrieron 256 eventos externos. Usando esta configuración, puede recibir interrupciones con intervalos de tiempo regulares siendo esto muy útil para cronometrar el tiempo, medir anchos de pulso, generar pulsos, y otras operaciones dependientes del tiempo.

La fuente del incremento del registro **RTCC** depende del bit 5 del registro **!option** (**RTS**). Si el bit es 0, el contador se incrementa con cada ciclo de instrucción. Si el bit es 1, el **RTCC** se incrementa cada vez que detecta un pulso en el pin **RTCC**. Usando el bit **RTE** (bit 4 de **!option**) puede determinar si el contador responde a flancos ascendentes (0) o descendentes (1).

Por defecto, el registro **RTCC** se incrementa con cada ciclo de instrucción o evento externo. A 50MHz, entonces, el **RTCC** requiere 20nS * 256 = 5.12uS para desbordarse cuando cuenta ciclos de instrucción. Este tiempo es demasiado corto para la mayoría de las aplicaciones (como verá pronto), así que a menudo deberá dividir el ciclo de reloj por algún valor. Puede hacer esto asignando el prescaler a **RTCC**. Este es el mismo prescaler que usa el temporizador watchdog, así que deberá asignarlo para un único uso. No puede escalar el contador **RTCC** y el temporizador watchdog a la vez.

Para asignar el prescaler a **RTCC**, borre el bit 3 del registro **!option** (**PSA**). Los últimos 3 bits del registro **!option** determinan el factor de división (ver Tabla VII.1). La relación máxima es 1:256 que a 50MHz se obtienen 1.3mS (.0013S). Por supuesto, si usa una frecuencia de reloj diferente también serán diferentes los tiempos. Obviamente, si usa una fuente externa en el pin **RTCC**, el tiempo entre desbordes dependerá de esa fuente.

Tip: Observe que la tabla no contiene la configuración 1:1. Esto es debido a que 1:1 es la que se obtiene cuando el prescaler está trabajando con el temporizador watchdog.

PS2	PS1	PS0	Relación	Tiempo de Desborde a 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)

Tabla VII.1 – Configuración del Prescaler

Pausas RTCC

Incluso sin interrupciones, el registro **RTCC** puede ser útil. En unidades anteriores, los programas usaban una pausa programada para retardar la ejecución de una instrucción. Si el **RTCC** se incrementa con el reloj de las instrucciones, puede emplearlo para cronometrar sus pausas fácilmente. Mire esta subrutina:

```
; se asume que el prescaler está en 1:256
pausa1_3ms          mov         rtcc,#1
; es correcto comparar con 0 porque el prescaler 256 está activado
:esperar            mov         w,rtcc
                    jnz         :esperar
                    ret
```

La subrutina pone a **rtcc** en 1 (que también, incidentalmente, limpia el prescaler). Luego espera a que **rtcc** sea igual a cero. Esto requerirá que cuente hasta 255 y cada incremento demanda 256 ciclos de instrucción. Por lo tanto, a 50MHz, la pausa total es 256*255*20nS = 1.3mS.

No olvide que al escribir en **rtcc** limpia el prescaler. Esto puede causar efectos secundarios. Por ejemplo, usted podría haber intentado usar la instrucción **test** para verificar si el valor del prescaler era cero. Esto no funcionará debido a que **test** mueve el registro sobre si mismo, modificando su valor. Mientras comprueba la igualdad a cero también borra el prescaler, de forma que el registro **rtcc** nunca se incrementa.

Otra posible fuente de error es la comparación con cero. Si no se activa el prescaler, **rtcc** se incrementa con cada ciclo de instrucción. Entonces, sería peligroso buscar la igualdad con un valor específico del prescaler. ¿Por qué? Por que **rtcc** podría asumir ese valor mientras está ejecutando otra instrucción. Por ejemplo, suponga que la subrutina anterior carga a **w** con $FF en el rótulo **:esperar**. Con el prescaler apagado, en la siguiente pasada por el bucle el contador valdrá 3, habiendo sido cero mientras ejecutaba la instrucción **jnz.**

Interrupciones RTCC

Para habilitar las interrupciones por desborde de **RTCC**, limpie el bit **RTI** (bit 6) en el registro **!option**. Una vez que este bit está en cero, el controlador detendrá lo que está haciendo cuando **RTCC** se desborde, y ejecutará el código que comienza en la dirección 0. Por supuesto, hasta ahora, su programa comenzaba en la dirección 0, pero eso se debía a que la directiva **reset** así lo indicaba. Puede comenzar su programa en otro lugar de la memoria para dejar lugar a la rutina de proceso de la interrupción.

Cuando ocurre una interrupción, el SX deshabilita la posibilidad de otras interrupciones. Además, almacena los registros **status**, **fsr** y **w**. El SX luego limpia los 3 bits superiores del registro **status** (estos bits son la parte superior de las direcciones de los saltos) y salta a la dirección 0. Todo este trabajo es necesario para que la rutina de atención a la interrupción (ISR) no interfiera con la ejecución del programa principal. Una vez finalizado el ISR, usa la instrucción **reti** para devolver el control al programa principal. Esto también habilita futuras interrupciones.

Tip: A diferencia de otros procesadores, el SX almacena los registros operativos (**w**, **fsr**, y **status**) en sectores temporarios especiales, y no en el stack. Sin embargo, el chip no atiende interrupciones que se presenten mientras esté atendiendo una interrupción previa.

Tal vez, la aplicación más simple de la interrupción **rtcc,** sea simular un reloj de tiempo real más grande. Recuerde que incluso con el prescaler máximo, el **rtcc** se desborda cada 1.3mS aproximadamente (a 50MHz). ¿Qué sucede si busca una pausa de 100mS? Por supuesto que puede llamar la rutina de pausa de 1.3mS unas 100 veces. Pero si tuviera un registro **rtcc** de 16-bits simplemente podría contar hasta 19531 (cada cuenta toma alrededor de 5uS con el prescaler en 1:256).

Este es un programa simple para hacer titilar LEDs cada 100mS basado en estas ideas:

```
            device      turbo,stackx_optionx
            reset       inicio
            freq        50000000   ; 50 Mhz

            org   8
rtcc1       ds    1

            org         0
isr         inc         rtcc1       ; manejo de interrupción
            reti

inicio
            mov         !rb,#$80    ; 7 salidas, 1 entrada
; configura RTCC con reloj interno y relación 1:256
            mov         !option,#$87
bucle       xor         rb,#$FF
            call        pausa100ms
            jmp         bucle

pausa100ms  clr         rtcc
            clr         rtcc1
:espera     mov         w,#$4c   ; $4c4b es 19531
            mov         w,rtcc1-w
            jnz         :espera
:espera0    mov         w,#$4b
            mov         w,rtcc-w
            jnz         :espera0
            ret
```

Interrupciones Periódicas

En los ejemplos anteriores, el programa principal hace parpadear un LED y controla la pausa. Sin embargo, el verdadero poder de las interrupciones se aprecia cuando le permite al ISR realizar una tarea, aparentemente mientras la rutina principal se está ejecutando. Mire este programa:

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

            org         0
isr         inc         rtcc1
            cjne        rtcc1,#$4D,isal  ; parpadea cada $4D00 períodos
            xor         rb,#$FF
; tiempo de reinicio
            clr         rtcc
            clr         rtcc1
isal
            reti

inicio
            mov         !rb,#$80    ; 7 salidas
; configura RTCC con reloj interno y relación 1:256
            mov         !option,#$87
bucle
            jmp         bucle
```

El programa principal configura **!rb**, **!option**, y luego ejecuta la instrucción **jmp** para detener el programa mientras ejecuta un bucle que no hace nada. Todo el trabajo se realiza en ISR. Es interesante notar que el ISR reinicia el registro **rtcc** de forma que la interrupción ocurrirá periódicamente. Esto no es inusual cuando se pretende que la interrupción se repita a intervalos regulares. Por supuesto, el intervalo será un poco mayor de lo que usted podría esperar.

Hay un problema con esto, sin embargo. Un ISR complejo puede tomar una cantidad de tiempo distinta de acuerdo a la situación actual. Esto puede ocasionar errores de temporización, intolerables en aplicaciones precisas. Por ejemplo, en el código anterior, la instrucción **reti** agrega una pequeña pausa al tiempo total, aunque para este caso carece de importancia.

Una respuesta mejor se obtiene al usar la instrucción **retiw** al final de ISR, específicamente cuando el prescaler esté apagado. Esta instrucción agrega el contenido del registro **w** a **rtcc**. Digamos que el controlador configura a **rtcc** para generar una interrupción cuando se desborde, y que el prescaler esté asignado al temporizador

watchdog. Cada incremento de **rtcc** representa 20nS (asumiendo como siempre, un reloj de 50MHz). Cuando comienza la interrupción el **rtcc** ya contó hasta 3. A medida que la ISR se ejecuta, el **rtcc** continúa incrementándose. Para determinar el tiempo con precisión, debe tener en cuenta cuánto tiempo ha pasado. Afortunadamente, hay una solución más simple; el registro **rtcc** ya tiene esa información. Si resta el número de ciclos que desea entre cada interrupción del número de ciclos que han transcurrido, obtendrá la cantidad exacta de ciclos requeridos.

Por ejemplo, digamos que quiere una interrupción cada 50 ciclos (1uS). Puede simplemente usar estas dos líneas de código al final de su ISR:

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

Lo único que hay que tener en cuenta es que su ISR, incluyendo los 3 ciclos de establecimiento de la interrupción, no debe exceder los 46 ciclos. Si lo hace, perderá la siguiente interrupción, o regresará al programa principal solamente para que ocurra otra interrupción inmediatamente. Debido al tiempo de establecimiento de la interrupción, siempre deberá dejar 3 ciclos, más el tiempo de ejecución de una instrucción en el programa principal, dando un total de 6 ciclos. Sin embargo, incluso con esta configuración su programa no se ejecutará muy seguido. Debería permitir una cantidad más generosa de tiempo entre interrupciones en la mayoría de los casos.

Un Ejemplo de Reloj

Una computadora que sabe qué hora es, puede ser muy útil. Podría servir para realizar la cuenta regresiva de un cohete de aeromodelismo, o cronometrar las lecturas de un sensor. Con una interrupción precisa es fácil controlar el tiempo. La parte difícil es traducir la rápida secuencia de interrupciones en números más significativos para los humanos. Este es un programa simple que usa un reloj de 50MHz con el registro **rtcc**. El ISR suma –al **rtcc** para generar una interrupción periódica cada 1uS. El ISR mantiene dos contadores de 16-bits registrando milisegundos y microsegundos.

Por supuesto, cada 1000 milisegundos obtenemos un segundo, cada 60 segundos tenemos un minuto, y 60 minutos determinan una hora. Puede extender esto fácilmente para contar días si lo desea. El programa principal en este caso no hace nada, pero podría agregar el código que crea necesario.

Este es un programa difícil de depurar debido a que si lo detiene mientras funciona, no muestra correctamente el tiempo. Puede ejecutar el programa a toda velocidad en el depurador y presionar el botón Poll para ver la variación en el tiempo. También verá titilar los LEDs en el puerto B y, si conecta un parlante piezoeléctrico a uno de los pines del puerto B, escuchará el reloj de su SX.

```
device      turbo,stackx_optionx
reset inicio
freq        50000000   ; 50 Mhz
org         8
```

```
microbajo   ds          1
microalto   ds          1
milibajo    ds          1
milialto    ds          1
segundos    ds          1
minutos     ds          1
horas       ds    1
            watch horas,8,udec
            watch minutos,8,udec
            watch segundos,8,udec

            org         0
isr         inc         microbajo
            snz
            inc         microalto
            cjne        microalto,#$03,salida_i ; titila cada $03e8
            cjne        microbajo,#$e8,salida_i ; períodos
; 1000 uS
            clr         microbajo
            clr         microalto
            inc         milibajo
            snz
            inc         milialto
            cjne        milialto,#$03,salida_i
            cjne        milibajo,#$e8,salida_i
; 1000 ms
            clr         milialto
            clr         milibajo
            xor         rb,#$FF  ; invierte LEDs
            inc         segundos
            cjne        segundos,#60,salida_i
; desborde segundos
            clr         segundos
            inc         minutos
            cjne        minutos,#60,salida_i
; desborde minutos
            clr         minutos
            inc         horas
            cjne        horas,#24,salida_i
; desborde horas
            clr         horas
; puede contar días si se quiere
```

```
; tiempo de reinicio
salida_i
            mov         w,#-50    ; interrupción cada 1uS
            retiw

inicio
            mov         !rb,#$00     ; todas salidas
            clr         microalto
            clr         microbajo
            clr         segundos
            clr         horas
            clr         minutos
; configura RTCC con reloj interno, relación 1:1
            mov         !option,#$88  ; sin prescaler
bucle
            jmp         bucle
```

Interrupciones Externas Mediante RTCC

Cuando quiere usar el pin **RTCC** para monitorear eventos externos, usualmente piensa en contar pulsos. Puede hacer esto, por supuesto. Cuando activa el bit 5 de **!option** (el bit **RTS**), el pin controla los pulsos incrementando **RTCC**. Si el bit **RTE** (bit 4 de **!option**) está en cero, el incremento se produce en los flancos ascendentes, de otra forma el SX detecta los flancos descendentes. El prescaler está disponible, así que puede usarse para dividir la cantidad de pulsos de entrada si se quiere.

Sin embargo, ¿qué sucede si desea una interrupción externa con un único pulso? La primera impresión es que esto no podría hacerse con **RTCC**. Después de todo, incluso con el prescaler asignado al watchdog, aún necesita 256 pulsos para obtener una interrupción, ¿correcto?

Mientras que esto es cierto, hay un truco para hacer que **RTCC** simule una interrupción externa. Simplemente cargue el registro **RTCC** con $FF. Asumiendo que el prescaler esté apagado y que el bit **RTS** esté en uno, el siguiente pulso de entrada causará una interrupción. Una técnica simple pero efectiva. Por supuesto, el ISR reiniciará **RTCC** en $FF antes de ejecutar la instrucción **reti,** para que la interrupción quede "armada" para el próximo evento.

Detección de Flancos de Entrada del Puerto B

Además del truco del **RTCC**, puede configurar cualquiera (o todos) los pines del puerto B como interrupciones externas. El puerto B tiene dos registros especiales que le permiten detectar flancos de entrada. Éstos se encuentran en funcionamiento todo el tiempo, no solamente cuando las interrupciones estén habilitadas. Como otros registros de puerto especiales, son accedidos mediante **!rb** cuando el registro **M** contiene cierto valor especial. Si **M** es $A, puede seleccionar el flanco que controlan los pines. Un bit en 1 hace que el SX detecte flancos descendentes (transiciones de 1 a 0) en ese pin. Un bit en 0 detecta flancos ascendentes o transiciones de 0 a 1. Cuando el flanco seleccionado aparece en un pin, el SX activa el bit correspondiente del registro de

detección de flancos de entrada (MIWU, **!rb** con **M** = $9). El SX nunca limpia este registro. Cuando su programa escribe el registro **W** en **!rb** y **M** es $9, el SX en realidad intercambia ambos valores. Así que puede leer el estado de los bits y limpiarlos a la vez.

Este proceso ocurre permanentemente. La mayoría de los programas simplemente ignoran esta característica. Sin embargo, puede ser usado para detectar la existencia de un flanco incluso cuando no esté usando las interrupciones del puerto B. Si conecta el circuito de la Figura VII.1 a varios pines del puerto B, puede probar este programa:

```
            device      turbo,stackx_optionx
            reset       inicio
            freq        50000000   ; 50 Mhz
            org         8
microbajo   ds          1
microalto   ds          1
milibajo    ds          1
milialto    ds          1
segundos    ds          1
minutos     ds          1
horas       ds          1
flancos     ds          1

            watch horas,8,udec
            watch minutos,8,udec
            watch segundos,8,udec

            org         0
isr         inc         microbajo
            snz
            inc         microalto
            cjne        microalto,#$03,salida_i       ; titila cada $03e8
            cjne        microbajo,#$e8,salida_i       ; períodos
; 1000 uS
            clr         microbajo
            clr         microalto
            inc         milibajo
            snz
            inc         milialto
            cjne        milialto,#$03,salida_i
            cjne        milibajo,#$e8,salida_i
; 1000 ms
            clr         milialto
```

```
            clr         milibajo
            inc         segundos
            cjne        segundos,#60,salida_i
; desborde segundos
            clr         segundos
            inc         minutos
            cjne        minutos,#60,salida_i
; desborde minutos
            clr         minutos
            inc         horas
            cjne        horas,#24,salida_i
; desborde horas
            clr         horas
; puede contar días si se quiere
; tiempo de reinicio
salida_i
            mov         w,#-50   ; interrupción cada 1uS
            retiw

inicio
            mov        !rb,#$FF
areset      clr         microalto
            clr         microbajo
            clr         segundos
            clr         horas
            clr         minutos
; configura RTCC con reloj interno, relación 1:1
            mov         !option,#$88  ; sin prescaler

; enciende los resistores de pull up del puerto B
            mode        $E
            mov         !rb,#$00
; configura al pin 0 del puerto B para interrupción con flanco descendente
            mode        $A    ; selecciona flanco
            mov         !rb,#$FF
            mode        $9    ; habilita interrupciones
            mov         !rb,#%0  ; limpia registro
; espera 10 segundos
espera10    cjne        segundos,#10,espera10
            mov         !rb,#%0  ; lee y limpia
            mov         flancos,w
```

```
; importante: reinicia el registro de modo
            mode        $F
            mov         !rb,#0    ; todas salidas
; invierte el sentido de los bits de flanco
            not         flancos
            mov         rb,flancos

bucle
; espera activa para que registre las entradas
            jmp         bucle
```

Este es más o menos el mismo programa que antes, pero no produce el parpadeo de los LEDs. En lugar de eso, espera durante 10 segundos (fácil de hacer con la rutina de interrupción de reloj) y luego enciende los LEDs que corresponden a los botones que se presionaron durante esos 10 segundos. Esto es muy simple usando la característica MIWU. Debido a que los LEDs se encienden cuando se coloca un 0 en el puerto de salida, el programa usa la instrucción **not** para invertir los bits leídos.

Tip: Este programa inicialmente configura el registro de dirección de forma que todos los pines del puerto B sean entradas. Luego, después de una pausa, pone a todos los pines como salidas. Un error muy común en este punto es olvidarse de configurar el registro **M** nuevamente a $F, antes de convertirlos en salidas. El código de detección de flanco modifica el valor de **M**, así que usted debe volverlo a $F antes de acceder el registro de dirección.

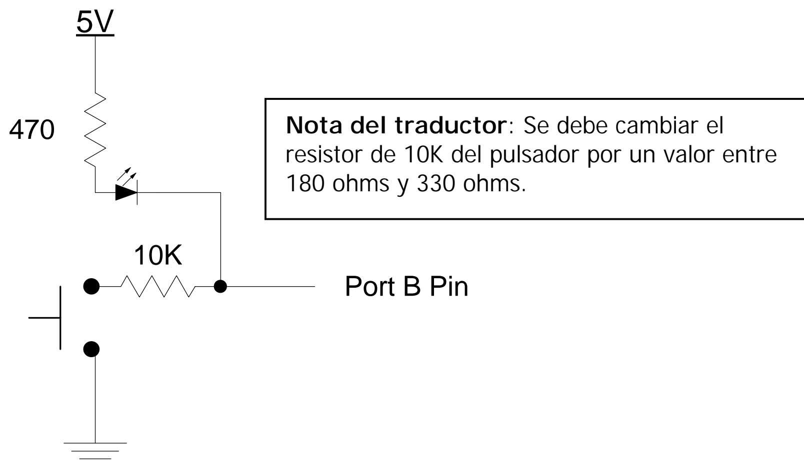

Figura VII.1 – Circuito Interruptor/LED

Interrupciones del Puerto B

Cuando el SX detecta un flanco, también puede generar una interrupción. Obtendrá esta configuración borrando los bits del registro **!rb** mientras **M** es igual a $B. Cuando el SX detecta un flanco en el pin correspondiente, generará una interrupción. Es trabajo de la ISR examinar el registro de pedido de interrupción y limpiarlo para que admita futuras interrupciones. Esta interrupción es igual que la de **rtcc,** almacena la información contextual del SX (registros especiales) y comienza en la ubicación 0.
Es posible usar las interrupciones del puerto B y **rtcc** a la vez, pero puede originar confusiones. Por ejemplo, si ocurre un pulso mientras se ejecuta la ISR, el SX no generará una interrupción al regresar de la ISR hasta que ocurra un nuevo evento. De igual modo, si **rtcc** se desborda mientras el SX está procesando una interrupción del puerto B, se perderá la interrupción **rtcc**. En algunos casos, la temporización no es crítica, así que perder un microsegundo o dos no es tan importante. Sin embargo, si se necesita mucha precisión en la temporización debería considerar trabajar con una sola fuente de interrupción (puerto B o **rtcc**) en el programa.

Tip: Si necesita un reloj de tiempo real y detección de flancos, piense en usar la interrupción **rtcc** a mayor velocidad, examinando los bits de pedido de interrupción luego de cada ejecución de la ISR (esto se conoce como *polling o escrutinio*). En muchas aplicaciones, es suficiente controlar las entradas cada microsegundo.

También es posible utilizar la interrupción del puerto B para reactivar el controlador después de una instrucción **sleep**. Si se presenta una interrupción del puerto B después de una instrucción **sleep**, no ocurre una interrupción. En su lugar, el controlador se reinicia con el bit 3 del registro **status** en cero y el bit 4 en uno. Esto no puede lograrse con una interrupción **rtcc**.

Resumen

Las interrupciones no deber ser necesariamente difíciles de usar. Esto es especialmente cierto en el SX debido a que el chip se encarga de muchos detalles por usted. Las interrupciones son esenciales cuando se deben procesar las entradas mientras se estaba ejecutando otra tarea, se debe controlar el tiempo, o se deben generar salidas específicas mientras realiza otras tareas.
Las interrupciones, junto con la gran velocidad del SX, forman las bases de la estrategia de periféricos virtuales que se tratan en la siguiente unidad. Aunque el trabajo con las interrupciones requiere un poco de diseño cuidadoso, y puede ser difícil de depurar, éstas valen lo que cuestan.

Ejercicios

1. Escriba un programa que utilice una interrupción por temporizador para medir (como mínimo) segundos. Normalmente, el programa no hace nada. Sin embargo, cuando presiona un botón conectado al pin 0 del puerto B, el programa encienda un LED (o haga tic tac con un parlante piezoeléctrico) cada segundo hasta que vuelva a presionar el botón. Al pulsar el botón por tercera vez el LED debe volver a titilar y así sucesivamente. Use la interrupción **rtcc** para la temporización y controle el interruptor en el programa principal.
2. Modifique el programa anterior de forma que la ISR controle el botón de entrada usando MIWU (detección por flanco), pero sin usar las interrupciones del puerto B.
3. Modifique el programa para que use ambas interrupciones; **rtcc** y puerto B.
4. ¿Cuál de los tres programas piensa usted que es la mejor aproximación?

Respuestas

1. La solución es directa. Observe que no puede usar la instrucción **sleep** porque el programa se detendría.

```
            device      turbo,stackx_optionx
            reset       inicio
            freq        50000000   ; 50 Mhz
            org         8
microbajo   ds          1
microalto   ds          1
milibajo    ds          1
milialto    ds          1
segundos    ds          1
tictac      ds          1
tmp         ds          1


            org         0
isr         inc         microbajo
            snz
            inc         microalto
            cjne        microalto,#$03,salida_i       ; titila cada $03e8
            cjne        microbajo,#$e8,salida_i       ; períodos
; 1000 uS
            clr         microbajo
            clr         microalto
            inc         milibajo
            snz
            inc         milialto
            cjne        milialto,#$03,salida_i
            cjne        milibajo,#$e8,salida_i
; 1000 ms!
            clr         milialto
            clr         milibajo
            test        tictac
            jz          notic
            xor         rb,#$FF   ; invierte LEDs
notic       inc         segundos
salida_i
            mov         w,#-50    ; interrupción cada 1uS
            retiw
```

7

```
inicio
            mov         !rb,#$01    ; 7 salidas, 1 entrada
            clr         microalto
            clr         microbajo
            clr         segundos
            clr         tictac
; configura RTCC con reloj interno relación 1:1
            mov         !option,#$88  ; sin prescaler

bucle
; espera activa para que registre las entradas
            jb          rb.0,bucle
; botón presionado
            not         tictac
; pausa antirebote (aproximadamente 1 segundo)
milbucle0   test        milialto   ; espera que milialto sea 0
            jnz         milbucle0
milbucle1   test        milialto
            jz          milbucle1  ; espera que no sea 0
milbucle    test        milialto
            jz          milbucle   ; espera que sea cero otra vez
            jmp         bucle
```

2. Comparado con el último programa, este tiene una ISR similar, pero un programa principal muy diferente (todo el trabajo se hace en la ISR). Observe que la ISR modifica el registro **M**, así que deberá guardarlo y recuperarlo para asegurarse que el registro **M** del programa principal no cambie (por supuesto, en este caso, el programa principal no tiene importancia, pero normalmente este no será el caso). Para eliminar el rebote, el código examina el registro de flancos cada 1mS.

```
            device      turbo,stackx_optionx
            reset       inicio
            freq        50000000    ; 50 Mhz
            org         8
microbajo   ds          1
microalto   ds          1
milibajo    ds          1
milialto    ds          1
segundos    ds          1
tictac      ds          1
tmp         ds          1

            org         0
isr
            inc         microbajo
            snz
            inc         microalto
            cjne        microalto,#$03,salida_i       ; titila cada $03e8
            cjne        microbajo,#$e8,salida_i       ; períodos
; 1000 uS
            clr         microbajo
            clr         microalto

; revisa el botón cada 1ms
            mov         tmp,M        ; guarda el registro M
            mode        $9
            clr         w
            mov         !rb,w        ; intercambia w y control de flancos
            and         w,#1         ; controla el bit bajo
            sz
            not         tictac       ; invierte tictac
            mov         M,tmp        ; recupera M

; milisegundos
            inc         milibajo
            snz
```

7

```
            inc         milialto
            cjne        milialto,#$03,salida_i
            cjne        milibajo,#$e8,salida_i
; 1000 ms!
            clr         milialto
            clr         milibajo
            test        tictac
            jz          notic
            xor         rb,#$FF      ; invierte LEDs
notic       inc         segundos
salida_i
            mov         w,#-50       ; interrupción cada 1uS
            retiw

inicio
            mov         !rb,#$01           ; 7 salidas
            clr         microalto
            clr         microbajo
            clr         segundos
            clr         tictac
; configura RTCC a reloj interno relación 1:1
            mov         !option,#$88       ; sin prescaler
; configura detección flanco descendente puerto B
            mode        $A                 ; selecciona flanco
            mov         !rb,#$FF

bucle
            jmp         bucle
```

3. Esta versión es, tal vez, la menos satisfactoria de las tres. Requiere interruptores que no reboten demasiado debido a que es difícil filtrar interrupciones múltiples causadas por el rebote. Además, si un evento **rtcc** ocurre cuando se procesa el accionamiento del botón, el tiempo se vuelve impreciso.

```
            device      turbo,stackx_optionx
            reset       inicio
            freq        50000000   ; 50 Mhz
            org         8
microbajo   ds          1
microalto   ds          1
milibajo    ds          1
milialto    ds          1
```

```
segundos    ds          1
tictac      ds          1
tmp         ds          1


            org         0
isr
; controla si se presionó el botón
            mov         tmp,M       ; almacena registro M
            mode        $9
            clr         w
            mov         !rb,w       ; intercambia w y control de flancos
            and         w,#1        ; controla bit bajo
            jz          rtccisr
            not         tictac      ; invierte tictac
            mov         M,tmp       ; restaura M
            iret

rtccisr
            mov         M,tmp
            inc         microbajo
            snz
            inc         microalto
            cjne        microalto,#$03,salida_i  ; titila cada $03e8
            cjne        microbajo,#$e8,salida_i  ; períodos
; 1000 uS
            clr         microbajo
            clr         microalto


; milisegundos

            inc         milibajo
            snz
            inc         milialto
            cjne        milialto,#$03,salida_i
            cjne        milibajo,#$e8,salida_i
; 1000 ms
            clr         milialto
            clr         milibajo
            test        tictac
            jz          notic
```

```
            xor         rb,#$FF       ; invierte LEDs
notic       inc         segundos
salida_i
            mov         w,#-50        ; interrupción cada 1uS
            retiw
inicio
            mov         !rb,#$01      ; 7 salidas
            clr         microalto
            clr         microbajo
            clr         segundos
            clr         tictac
; configura RTCC con reloj interno relación 1:1
            mov         !option,#$88        ; sin prescaler
; configura detección de flanco descendente en puerto B
            mode        $A                  ; selecciona flanco
            mov         !rb,#$FF
            mode        $B                  ; habilita interrupción en pin 0
            mov         !rb,#$FE

bucle
            jmp         bucle
```

4. Es muy claro que el programa 3 requerirá mucho trabajo para hacerlo robusto. Mezclar dos fuentes de interrupción es un trabajo peligroso. Sobre las otras dos técnicas, es cuestión de gustos personales. El código 1 tiene más partes del programa en el bucle principal, donde serán más fáciles de depurar. Sin embargo, el 2 es bastante limpio y mantiene el proceso fuera del programa principal (presumiblemente, realizará alguna tarea en el programa principal).

Los programas y la información de este texto son para propósitos educativos, habiendo sido cuidadosamente probados, pero no están garantizados para ningún propósito en particular. El editor, el autor y el traductor no ofrecen ninguna garantía sobre la precisión, adecuación, o completitud de la información presentada, y no asumen ninguna responsabilidad por errores u omisiones en el texto. Tampoco asumen la responsabilidad por daños emergentes por el uso de la información de este texto, o por cualquier violación de los derechos intelectuales o de autor de terceros, que resulten del uso de esta información.

Versión en Castellano 1.0

Unidad VIII. Periféricos Virtuales

Unidad VIII de Introducción al Lenguaje Assembler con el Microcontrolador Scenix SX

© 1999 Parallax, Inc. Reservados todos los derechos. Autor Al Williams, AWC. Traducción Arístides Álvarez.

La mayoría, o todos, los microcontroladores son valiosos porque se comunican con el mundo exterior de alguna forma. Como resultado, los diseñadores de sistemas pasaban mucho tiempo conectando los microcontroladores al mundo exterior. Con estos procesadores antiguos, se necesitaban componentes electrónicos adicionales para cada tarea. ¿Quiere leer tensión? Coloque un chip A/D (conversor analógico a digital). ¿Quiere comunicarse con una PC? Coloque un chip UART (Transmisor y Receptor Asincrónico Universal).

En los últimos años, los fabricantes de microcontroladores han estado integrando chips de periféricos comunes directamente dentro del microcontrolador. Esto le permite diseñar sistemas de forma más simple y conserva la capacidad de E/S del microcontrolador. El único problema es que ningún microcontrolador puede incorporar todos los periféricos posibles. Para un proyecto puede necesitar un UART. El siguiente proyecto puede requerir dos entradas A/D. Otro proyecto puede necesitar una A/D y dos UARTs. Obviamente, no importa que tan brillantes sean los diseñadores del microcontrolador, usted nunca tendrá todos los periféricos que necesite en el interior del microcontrolador.

Otro problema de esta aproximación es que deberá usar diferentes microcontroladores para diferentes tareas. No podrá usar un microcontrolador con entradas A/D en el lugar de uno con UART. Esto causará que deba trabajar con muchos modelos de microcontroladores. Idealmente, usted preferirá usar el mismo componente en todos sus diseños. Por lo menos, intentará usar la menor variedad posible de componentes.

Scenix resuelve este problema con los *Virtual Peripherals (Periféricos Virtuales)* o VPs. Los VPs sacan ventaja de la gran velocidad del SX y la capacidad de interrupciones para simular dispositivos periféricos tradicionales en software en lugar de implementarlos en hardware. Esto tiene muchas ventajas:

1) Usa el mismo componente en todos los diseños
2) Puede agregar los dispositivos que necesite en cada proyecto en particular
3) Puede modificar los dispositivos según su necesidad (normalmente imposible en el hardware)

Un VP es simplemente un módulo de código (usualmente una rutina de atención de interrupción o ISR) que simula un dispositivo de E/S. Puede descargar muchos VPs desde el sitio Web de Scenix (www.scenix.com). Otros VPs pueden encontrarse en otros sitios (gratis o no). Incluso usted puede escribir sus propios VPs para usar en proyectos futuros o para vender a otros programadores. Algunos VPs necesitarán algunos componentes externos (usualmente algunos capacitores o resistores). Otros trabajan completamente en software.

Usando un VP

Cuando comienza el diseño de un proyecto con el SX, primero debería fijarse si hay algún VP estándar que le sea de utilidad. Scenix genera frecuentemente nuevos VPs, pero estos son unos de los más útiles que se encuentran disponibles en el momento de escribir este libro:

- DTMF Generation – Genera tonos telefónicos
- FSK Detection – Recibe datos FSK (desplazamiento de fase)
- FSK Generation – Emite datos FSK
- I2C – Interfaz con chips con bus IIC (un VP para esclavo, otro para maestro)
- SPI – Interfaz con chips con bus SPI (un VP para esclavo, otro para maestro)
- UART – E/S serial (hasta 230.4 Kbaud)
- Multi UART – 8 puertos seriales, cada uno a 19.2 Kbaud
- LCD – Maneja un módulo LCD Hitachi estándar (un VP para 4 bits, otro para 8 bits)
- LED – Maneja LEDs de siete segmentos
- PWM – Una variedad de VPs que le permiten generar modulación por ancho de pulso, útil para generar tensiones, controlar la velocidad de motores y tareas similares
- ADC – Usando unos pocos componentes puede hacer un ADC, casi completamente en software
- Stepper Motor – Controla motores paso a paso
- Timers –VPs comunes que pueden implementas temporizadores y relojes de tiempo real
- Input – VPs para eliminar el rebote de botones y controlar teclados

Tip: Asegúrese de revisar la última lista en http://www.scenix.com/virtual/vp/sx_library_5.pdf.

Una vez escogido el VP, debe integrarlo a su programa. Podría estar interesado en usar más de un VP. Puede hacer esto (ver más adelante), pero por ahora usemos solamente uno. Como ejemplo, suponga que quería construir un circuito que marque el número de teléfono de Parallax usando DTMF usando un parlante piezoeléctrico conectado al pin 6 del puerto C.

Si mira la página Web de Scenix, verá que hay un archivo de texto que describe el VP de generación de DTMF y suministra el código para un programa de ejemplo. El problema con los programas de ejemplo que se normalmente hacen cosas que usted no necesita, así que tendrá que cortar y pegar las piezas que desee en su programa.

El programa de ejemplo lee datos de un puerto RS-232 y marca el número que se le ordene. Para este ejemplo, no necesitará el VP de E/S serial. Sin embargo, dando un vistazo a la ISR del ejemplo, observamos que también se encuentran los VPs de PWM y temporización. Un control detallado revela que ambos son necesarios para el VP de DTMF.

Además de la ISR, también deberá incorporar las variables que usan las rutinas y varias subrutinas que le ayudan a llamar a las funciones VP. El VP también puede necesitar una inicialización específica de los registros

de control de puertos, el registro **!option**, o variables internas. Al final deberá recurrir al ensayo de prueba y error, salvo que está preparado para comprender completamente lo que hace el programa.

Una vez que crea tener todo lo necesario, querrá usar el comando Run | Assemble para ver si obtiene errores de compilación. Si no tiene errores, el programa puede estar listo (aunque aún puede conservar errores que no sean sintácticos).

A menudo, el VP no usa la misma asignación de puertos que usted pretendía. Usualmente puede intercambiar los números de los pines sin efectos perjudiciales. Sin embargo, sea cuidadoso. Si el VP está usando, por ejemplo, la capacidad de interrupciones del puerto B, no podrá mover los pines al puerto A o C debido a que no tienen interrupciones. Usualmente el VP tiene una declaración al principio que fija las E/S (**PWM_pin**, en este caso). Además de modificar esta declaración, deberá encontrar todos los lugares donde el VP hace referencia a los registros **ra**, **rb**, **rc**, **!ra**, **!rb**, o **!rc** y modificar esas líneas también.

Con el VP en su lugar, el programa principal es bastante simple:

```
; carga dígitos
            clr         i
digbucle    call        obtdigito
            mov         byte,w
            cje         byte,#$FF,listo
            call        @load_frequencies       ; rutina VP
            call        @dial_it                ; rutina VP
            inc         i
            mov         w,#20
            call        @delay_10n_ms
            jmp         digbucle
listo
            sleep
```

Para marcar nuevamente, reinicie el microcontrolador. Las rutinas **load_frequencies**, **dial_it**, y **delay_10n_ms** son parte del VP (y se encuentran en páginas diferentes lo que explica el prefijo @). La rutina **obtdigito** simplemente recorre una tabla que contiene el número telefónico.

Mezclando VPs

Cuando necesite mezclar VPs, hay varios temas que debe considerar:

1. ¿A qué frecuencia se debe ejecutar la ISR?
2. Conflictos de puertos y variables
3. Conflictos en el uso del registro **!option**
4. Rutas de tiempo variable a través de la ISR

La mayoría de las soluciones son directas. Algunas veces puede ajustar parámetros para resolver conflictos. Por ejemplo, si necesita una UART, puede ajustar su temporización para que pueda trabajar con otros VPs que no

usan la misma frecuencia. Otras soluciones son más difíciles y requieren un esfuerzo mayor para rescribir el código de los VPs.

Otro tema son las variaciones de tiempo para distintos caminos de la ISR. Algunos VPs dependen de una cantidad exacta de tiempo entre interrupciones. La generación de PWM, por ejemplo, requiere una temporización precisa. Si mezcla un VP que requiere una cantidad exacta de tiempo entre interrupciones con otro VP, debería colocar el código de la interrupción del VP sensible a las variaciones de tiempo antes que el código del otro VP. Invertir este orden causará un mal funcionamiento del VP sensible a los cambios de tiempo si la ISR del otro VP no se ejecuta siempre en el mismo tiempo. Algunos VPs usan técnicas especiales para asegurarse de obtener siempre el mismo tiempo de ejecución, pero la mayoría varía estos tiempos de acuerdo a las condiciones.

Resumen

Usando VPs puede crear programas poderosos con facilidad. Sin embargo, se necesita un poco de experiencia y esfuerzo para separar las partes necesarias de los ejemplos de VP y aplicarlas a su programa. El esfuerzo, sin embargo, normalmente es mucho menor que el necesario para llevar a cabo estas funciones creando su propio hardware o software.

Puede mezclar VPs si lo hace con cuidado. Sin embargo, mezclar VPs puede originar muchos conflictos debidos a los requerimientos de cada módulo.

Ejercicios

1. Descargue el VP de generación de DTMF y quite las partes que no sean necesarias para construir un programa que marque un número automáticamente cuando se inicia.
2. Mueva la salida del DTMF al pin 6 del puerto C.
3. Personalice el código para que marque el número que usted elija cada vez que se reinicie. Ponga a dormir (sleep) el controlador una vez marcado el número. Para escuchar los tonos, puede conectar un parlante piezoeléctrico en el puerto. Sin embargo, esta señal puede ser muy ruidosa y muy débil para marcar realmente el número. Si realmente quiere marcar el número en una línea telefónica, agregue un filtro RC (ver instrucciones en la documentación del VP; necesitará un resistor de 600 ohm y un capacitor de aproximadamente .2uF). También puede usar un parlante amplificado o un trazador de señal para aumentar el volumen.

Respuestas

Este es el listado del programa que cumple con los tres enunciados de esta unidad:
(Nota del Traductor: los nombres de los rótulos no serán traducidos debido a que vienen en el VP, sí se traducirán los comentarios respectivos)

```
            device      sx28l,stackx_optionx
            device      oscxt5,turbo

            freq  50_000_000             ; velocidad de ejecución = 50MHz
            ID    'DIAL'

            reset inicio                       ; inicio en caso de reset


;**************
; Declaraciones de datos comunes y frecuencias
;**************
f697_h      equ         $012  ; Frecuencia DTMF
f697_l      equ         $09d

f770_h      equ         $014  ; Frecuencia DTMF
f770_l      equ         $090

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

f941_h      equ         $019  ; Frecuencia DTMF
f941_l      equ         $021

f1209_h     equ         $020  ; Frecuencia DTMF
f1209_l     equ         $049

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

f1477_h     equ         $027  ; Frecuencia DTMF
f1477_l     equ         $071

f1633_h     equ         $02b  ; Frecuencia DTMF
f1633_l     equ         $09c
```

```
;**************
; Definiciones de pines
;**************
;PWM_pin          equ   rb.7        ; salida DTMF
PWM_pin           equ   rc.6        ; salida DTMF

;*************
;     Variables Globales
;*************
                  org   $8          ; registros Globales

flags             ds    1
dtmf_gen_en       equ   flags.1     ; dice si la salida DTMF está activada
timer_flag        equ   flags.2     ; indica desbordes de temporizadores
temp              ds    1           ; registro de almacenamiento temporal
byte              ds    1           ; un byte
i                 ds    1           ; contador del bucle

;*************
;     Variables del banco 0
;*************
                  org   $10

sin_gen_bank      =     $

freq_acc_high     ds    1           ;
; acumulador de 16-bit que decide cuándo incrementar la sinusoide
freq_acc_low      ds    1
freq_acc_high2    ds    1           ;
; acumulador de 16-bit que decide cuándo incrementar la sinusoide
freq_acc_low2     ds    1
freq_count_high   ds    1           ; freq_count = Frecuencia * 6.83671552
freq_count_low    ds    1           ; contador de 16-bit que
; decide la frecuencia de la sinusoide

freq_count_high2  ds    1           ; freq_count = Frecuencia * 6.83671552
freq_count_low2   ds    1           ; contador de 16-bit que
; decide la frecuencia de la sinusoide

curr_sin          ds    1           ; valor actual de la sinusoide
sinvel            ds    1           ; velocidad de la sinusoide
```

```
curr_sin2          ds    1      ; valor actual de la sinusoide
sinvel2            ds    1      ; velocidad de la sinusoide

sin2_temp          ds    1      ; usada para almacenamiento temporal

PWM_bank           =     $

pwm0_acc           ds    1            ; acumulador PWM
pwm0               ds    1            ; salida actual de PWM

;*************
;     Variables del banco 1
;*************
                   org     $30
timers             =     $
timer_l            ds    1
timer_h            ds    1

;**************
; Interrupción
;
; con un valor retiw de -163 y una frecuencia de 50MHz, este código
; se ejecuta cada 3,26us.
;**************
            org   0
;**************
PWM_OUTPUT
; emite el valor actual de pwm0 por el pin PWM_pin.  Genera una tensión
analógica en PWM_pin luego de filtrada
;**************
            bank  PWM_bank
            add   pwm0_acc,pwm0     ; suma la salida PWM al acc
            snc
            jmp   :carry            ; si no hay acarreo, borra
                                    ; el pin PWM_pin
            clrb  PWM_pin
            jmp   PWM_out
:carry
            setb  PWM_pin           ; caso contrario activa PWM_pin
PWM_out
;**************
```

```
            jnb   dtmf_gen_en,sine_gen_out
            call  @sine_generator1
sine_gen_out


;**************
do_timers
; El temporizador correrá a la velocidad de la interrupción (3.26us para
; 50MHz).  Para calibrar los temporizadores, modifique FFFFh -
; (valor que corresponde al tiempo.)
; Ejemplo:
; para 1ms = 1ms/3.26us = 306 dec = 132 hex así que haga $FFFF - $0132 =
; $FECD
;**************

            bank  timers            ; cambia al banco de temporizadores
            mov   w,#1
            add   timer_l,w         ; suma 1 a timer_l
            jnc   :timer_out        ; si no es cero, entonces
            add   timer_h,w         ; no incrementa timer_h
            snc
            setb  timer_flag
:timer_out
;**************
:ISR_LISTO
; Este es el fin de la rutina de atención a la interrupción
; Ahora carga 163 en w y ejecuta retiw
; para que la interrupción se produzca 163 ciclos después de inicio
; (3.26us@50MHz)
;**************
            break
; interrupción 163 ciclos después de esta detención
            mov   w,#-163
            retiw                   ; regresa de la interrupción
;**************

inicio      bank  sin_gen_bank      ; inicio del programa

      ;******************************************************
      ; Declaración de puertos y registros
      ;******************************************************
```

```
; use estos valores para una onda que está desfasada 90 grados.
            mov   curr_sin,#-4
            mov   sinvel,#-8
; use estos valores para una onda que está desfasada 90 grados.
            mov   curr_sin2,#-4
            mov   sinvel2,#-8
            call  @disable_o

            mov   !option,#%00011111      ; habilita interrupciones wreg y
            mov   !rc,#%10111111          ; rtcc

            mov   m,#$D                   ; configura niveles cmos
            mov   !rc,#%10111111
            mov   m,#$F

; carga dígitos
            clr   i
digbucle    call  getdigit
            mov   byte,w
            cje   byte,#$FF,listo
            call  @load_frequencies       ; carga registros de frecuencia
            call  @dial_it                ; marca el número por 60ms
; y regresa
            inc   i
            mov   w,#20
            call  @pausa_10n_ms
            jmp   digbucle
listo
            sleep

; obtiene i'ésimo dígito a marcar
getdigit    mov   w,i
            jmp   PC+W
            retw  1,8,8,8,5,1,2,1,0,2,4,$FF



org   $200              ; inicia el código en la página 1
;**********************************************************
;     Subrutinas varias
;**********************************************************
```

```
pausa_10n_ms
; Esta subrutina genera una pausa de 'w'*10 milisegundos.
; Esta subrutina usa el registro TEMP
; ENTRADA         w     -     número de milisegundos de pausa
; SALIDA    regresa después de n milisegundos.
;*******************************************************
            mov   temp,w
            bank  timers
:bucle      clrb  timer_flag        ; este bucle genera una pausa de 10ms
            mov   timer_h,#$0f4
            mov   timer_l,#$004
            jnb   timer_flag,$
            dec   temp              ; repite w-1 veces.
            jnz   :bucle
            clrb  timer_flag
            retp

;*******************************************************
; Subrutina - Desconectar las salidas
; Carga el valor de CC en PWM y desconecta el interruptor de salida.
;*******************************************************
disable_o
            bank  PWM_bank    ; modo de entrada.
            mov   pwm0,#128   ; pone 2.5V CC en el pin de salida PWM
            retp

org   $400                    ; esta tabla está en la página 2.
; tabla de tonos DTMF
_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              ; estas subrutinas están en la página 3.
```

```
;************************************
; subrutinas de transmisión de DTMF
;************************************
;************************************
load_frequencies
; Esta subrutina carga las frecuencias usando la tabla
; El índice de la tabla es pasado en el registro byte.  La tabla DTMF
; debe estar entre $400 y $500.
;************************************
            cje   byte,#$0FF,:end_load_it
            clc
            rl    byte
            rl    byte              ; multiplica el byte por 4 (offset)
            add   byte,#_0_         ; suma el offset del primer dígito
            mov   temp,#4
            mov   fsr,#freq_count_high

:dtmf_load_ bucle mov   m,#4              ; mueva 4 a m (tabla en $400)
            mov   w,byte
            IREAD                         ; obtiene el valor de la tabla
            bank  sin_gen_bank            ; y lo carga en el registro
            mov   indf,w                  ; frecuencia
            inc   byte
            inc   fsr
            decsz temp
            jmp   :dtmf_load_bucle  ; cuando cargó los 4 valores,
:end_load_it  retp                  ; regresa
;*************************************************
dial_it     ; Esta subrutina emite las frecuencias que se cargaron durante
            ; 1000ms, y luego se detiene.
;***************
            cje   byte,#$0FF,end_dial_it
            bank  sin_gen_bank
; use estos valores para iniciar onda en un valor cercano a cruce por cero
            mov   curr_sin,#-4
            mov   sinvel,#-8
; use estos valores para iniciar onda en un valor cercano a cruce por cero
            mov   curr_sin2,#-4
            mov   sinvel2,#-8
            enable_o                      ; habilita la salida
            mov   w,#3
            call  @pausa_10n_ms           ; pausa 30ms
```

8

```
            setb  dtmf_gen_en
            mov   w,#10
            call  @pausa_10n_ms           ; pausa 100ms
            clrb  dtmf_gen_en
            call  @disable_o              ; desconecta las salidas
end_dial_it retp
;**************************************************
sine_generator1               ;(parte de rutina de atención de interrup.)
; Esta rutina genera una sinusoide sintética con valores entre
; -32 y 32.  La frecuencia es especificada por el contador.  Para fijar la
; frecuencia, ponga este valor en el registro de 16-bits freq_count:
; freq_count = FRECUENCIA * 6.83671552 (@50MHz)
;****************************************************
            bank  sin_gen_bank
; genera sinusoide a frecuencia
            add   freq_acc_low,freq_count_low;2
            jnc   :no_carry         ;2,4  ; si el byte inferior desborda
            inc   freq_acc_high           ; suma el acarreo al superior
            jnz   :no_carry               ; si el acarreo causa desborde
; entonces suma el contador al acumulador (que debería ser cero,
; así que mov servirá)
            mov   freq_acc_high,freq_count_high
                                          ; y actualiza la sinusoide
            jmp   :change_sin
:no_carry
; suma los bytes superiores de los acumuladores
            add   freq_acc_high,freq_count_high
            jnc   :no_change
:change_sin

            mov   w,++sinvel        ;1    ; si la sinusoide
            sb    curr_sin.7        ;1    ; es positiva, desacelera
            mov   w,--sinvel        ;1    ; caso contrario, acelera.
            mov   sinvel,w          ;1
            add   curr_sin,w        ;1    ; suma la velocidad a la
                                          ; sinusoide


:no_change

;***************************************************
sine_generator2               ;( parte de rutina de atención de interrup.)
```

```
; Esta rutina genera una sinusoide sintética con valores entre
; -32 y 32.  La frecuencia es especificada por el contador.  Para fijar la
; frecuencia, ponga este valor en el registro de 16-bits freq_count:
; freq_count = FRECUENCIA * 6.83671552 (@50MHz)
;*************************************************

; genera sinusoide a frecuencia
            add   freq_acc_low2,freq_count_low2 ;2
            jnc   :no_carry         ;2,4  ; si el byte inferior desborda
            inc   freq_acc_high2    ; suma el acarreo al superior
            jnz   :no_carry               ; si el acarreo causa desborde
; entonces suma el contador al acumulador (que debería ser cero,
; así que mov servirá)
            mov   freq_acc_high2,freq_count_high2
                                          ; y actualiza la sinusoide
            jmp   :change_sin
:no_carry
; suma los bytes superiores de los acumuladores

            add   freq_acc_high2,freq_count_high2
            jnc   :no_change
:change_sin

            mov   w,++sinvel2 ;1          ; si la sinusoide
            sb    curr_sin2.7 ;1          ; es positiva, desacelera
            mov   w,--sinvel2 ;1          ; caso contrario, acelera.
            mov   sinvel2,w   ;1
            add   curr_sin2,w ;1          ; suma la velocidad a la
                                          ; sinusoide


:no_change
            mov   pwm0,curr_sin2    ; mov sin2 a pwm0
            mov   sin2_temp,w
; mueve el valor actual de la sinusoide de alta frecuencia
            clc                            ; a un registro temporario

; divide el registro temporario por 4 desplazando a la derecha
            snb   sin2_temp.7
            stc                     ; (para resultado = (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
; suma el valor de SIN a la salida PWM
            add   pwm0,curr_sin
; para el resultado  pwm0 = 1.25*sin2 + 1*sin
; pone pwm0 en la mitad del rango de salida (para no lidiar con valores
; negativos)
            add   pwm0,#128
            retp              ; regresa con los bits de la página intactos
```

Los programas y la información de este texto son para propósitos educativos, habiendo sido cuidadosamente probados, pero no están garantizados para ningún propósito en particular. El editor, el autor y el traductor no ofrecen ninguna garantía sobre la precisión, adecuación, o completitud de la información presentada, y no asumen ninguna responsabilidad por errores u omisiones en el texto. Tampoco asumen la responsabilidad por daños emergentes por el uso de la información de este texto, o por cualquier violación de los derechos intelectuales o de autor de terceros, que resulten del uso de esta información.

Versión en Castellano 1.0

Apéndice A. Resumen de Instrucciones

Apéndice A de Introducción al Lenguaje Assembler con el Microcontrolador Scenix SX

© 1999 Parallax, Inc. Reservados todos los derechos. Autor Al Williams, AWC. Traducción Arístides Álvarez.

Instrucciones de Control

Instrucción	Palabras	Ciclos Turbo	Descripción
BANK x	1	1	Configura el banco del registro actual
MODE x	1	1	Configura el modo de E/S
NOP	1	1	Ninguna operación
PAGE	1	1	Configura la página de código actual
SLEEP	1	1	Pone al Controlador en modo de bajo consumo

Control de Flujo

Instrucción	Palabras	Ciclos Turbo	Descripción
CALL	1	3	Llama una subrutina
CJA	4	4,6	Compara y salta si es mayor
CJAE	4	4,6	Compara y salta si es mayor o igual
CJB	4	4,6	Compara y salta si es menor
CJBE	4	4,6	Compara y salta si es menor o igual
CJE	4	4,6	Compara y salta si es igual
CJNE	4	4,6	Compara y salta si es distinto
CSA	3	3,4	Compara y saltea si es mayor
CSAE	3	3,4	Compara y saltea si es mayor o igual
CSB	3	3,4	Compara y saltea si es menor
CSBE	3	3,4	Compara y saltea si es menor o igual
CSE	3	3,4	Compara y saltea si es igual
CSNE	3	3,4	Compara y saltea si es distinto
DECSZ	1	1,2	Decrementa y saltea si es cero
DJNZ	2	2,4	Decrementa y salta si no es cero
INCSZ	1	1,2	Incrementa y saltea si es cero
IJNZ	2	2,4	Incrementa y salta si no es cero
JB	2	2,4	Salta si el bit es uno
JC	2	2,4	Salta si el acarreo es uno
JMP	1	3	Salta
JNB	2	2,4	Salta si el bit es cero
JNC	2	2,4	Salta si el acarreo es cero
JNZ	2	2,4	Salta si no es cero
JZ	2	2,4	Salta si es cero
MOVSZ	1	1,2	Mueve (con inc/dec opcional) y saltea si es cero
RET	1	3	Regresa de la subrutina
RETP	1	3	Regresa desde otra página
RETW	1	3	Regresa un valor
SKIP	1	2	Saltea la instrucción siguiente
SNB	1	1,2	Saltea si el bit es cero
SNC	1	1,2	Saltea si el acarreo es cero
SNZ	1	1,2	Saltea si no es cero

Matemática y Lógica

Instrucción	Palabras	Ciclos Turbo	Descripción
ADD	1	1	Suma (registro + W o W + registro)
ADD	2	2	Suma (registro + registro o literal)
ADDB	2	2	Suma un bit
AND	1	1	Y (registro y W, W y registro, W y literal)
AND	2	2	Y (registro y literal o registro y registro)
DEC	1	1	Decrementa
INC	1	1	Incrementa
NOT	1	1	Invierte
OR	1	1	O (registro y W, W y registro o W y literal)
RL	1	1	Rota a la izquierda
RR	1	1	Rota a la derecha
SUB	1	1	Resta W de un registro
SUB	2	2	Resta un registro de otro o literal de un registro
XOR	1	1	O exclusiva de registro y W o W y registro
XOR	2	2	O exclusiva de registro y registro o registro y literal

Manejo de Interrupciones

Instrucción	Palabras	Ciclos Turbo	Descripción
RETI	1	3	Regresa de interrupción
RETIW	1	3	Regresa de interrupción y suma W a rtcc

Manipulación de Bits

Instrucción	Palabras	Ciclos Turbo	Descripción
CLC	1	1	Borra el acarreo
CLRB	1	1	Borra un bit
CLZ	1	1	Borra la bandera de cero
MOVB	4	4	Mueve un bit
SETB	1	1	Activa un bit (pone en 1)
STC	1	1	Activa el acarreo
STZ	1	1	Activa la bandera de cero

Mover/Borrar/Verificar

Instrucción	Palabras	Ciclos Turbo	Descripción
CLR	1	1	Borra un registro, W, o WDT
MOV	1	1	Mueve W a registro, registro a W, literal a W
MOV	2	2	Mueve registro a registro o literal a registro
TEST	1	1	Verifica W o registro y configura banderas

Instrucciones Varias

Instrucción	Palabras	Ciclos Turbo	Descripción
IREAD	1	4	Lee la memoria de programa
LCALL	1-4	3-6	Obsoleta
LJMP	1-4	3-6	Obsoleta
LSET	0-3	0-3	Obsoleta

Los programas y la información de este texto son para propósitos educativos, habiendo sido cuidadosamente probados, pero no están garantizados para ningún propósito en particular. El editor, el autor y el traductor no ofrecen ninguna garantía sobre la precisión, adecuación, o completitud de la información presentada, y no asumen ninguna responsabilidad por errores u omisiones en el texto. Tampoco asumen la responsabilidad por daños emergentes por el uso de la información de este texto, o por cualquier violación de los derechos intelectuales o de autor de terceros, que resulten del uso de esta información.

Versión en Castellano 1.0

Apéndice B. Hardware

Apéndice B de Introducción al Lenguaje Assembler con el Microcontrolador Scenix SX

© 1999 Parallax, Inc. Reservados todos los derechos. Autor Al Williams, AWC. Traducción Arístides Álvarez.

Los proyectos de este libro son fáciles de construir usando componentes comunes. Para lograr la máxima flexibilidad, le convendrá usar una protoboard. Si usa la plaqueta SX-Tech de Parallax simplemente conecte los circuitos a la protoboard integrada.

También puede usa su propia protoboard. El chip SX simplemente requiere una fuente regulada de 5 volts (una fuente regulada común servirá) y una conexión al programador SX-Key. Si está usando el SX-Blitz, o quiere operar el circuito sin el SX-Key, necesitará además un resonador cerámico de 50MHz (Murata CST50.00MXW040 o equivalente).

Para completar exitosamente los ejercicios del libro, necesitará unos pocos componentes comunes:

- LEDs (o LEDs de 5V con resistores integrados)
- Resistores de 470 ohm (si no usa LEDs de 5V)
- Pulsadores
- Resistores de pull up (10K a 22K, 1/4W o 1/8W)
- Un parlante piezoeléctrico

Circuito Común

Todos los circuitos usan el SX conectado al programador y la circuitería de soporte. Nuevamente, si está usando la plaqueta SX-Tech ya tiene todo resuelto. Si está usando el SX-Key, solamente necesita conectar el chip a 5V, masa, y al SX-Key. Puede usar cualquier fuente de alimentación de 5V que posea (asegúrese de que sea regulada). Si quiere construir una fuente simple de 5V, vea la figura B.1. Esta fuente puede suministrar 100mA directamente, o hasta 1A si usa un 7805 con disipador en lugar del 78L05. Puede usar cualquier transformador de pared para alimentar la entrada de CC de este circuito.

Para asegurar un buen funcionamiento, también debería conectar el pin MCLR a 5V directamente o a través de un resistor de pull up. Si usa este resistor podrá conectar el pin MCLR a masa para reiniciar el controlador. Para lograr esto debería conectar un pulsador a masa.

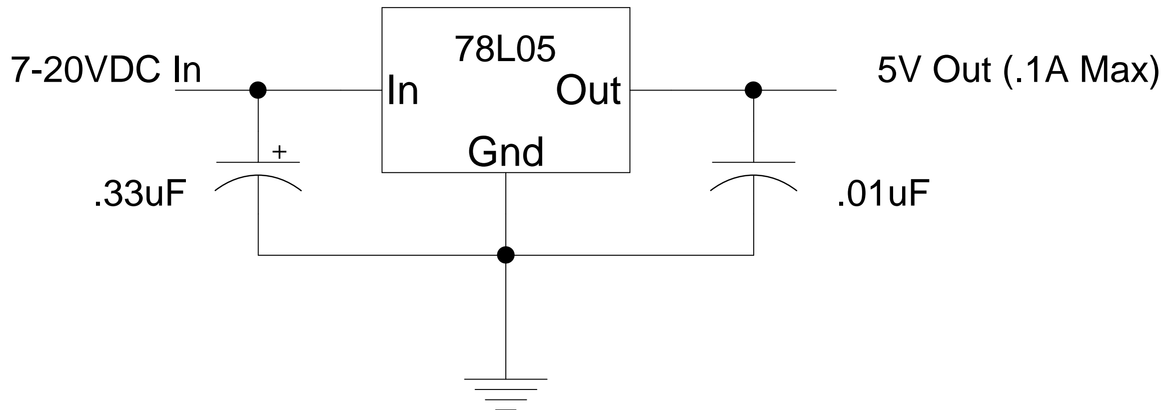

Figura B.1 Una Fuente de 5V Simple

Para conectar el programador, puede usar terminales separados una décima de pulgada (0,25 cm). Estos terminales son de venta común en negocios de electrónica. Inserte un extremo en la protoboard y el SX-Key (o SX-Blitz) en el otro. Si un lado de los pines es demasiado corto, normalmente puede deslizar el espaciador de plástico hasta lograr la misma longitud de cada lado. La Tabla B.1 muestra las conexiones necesarias.

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

Tabla B.1 Conexiones de los pines del SX

Circuitos de E/S

La mayoría de los proyectos del libro usan alguna entrada o salida. La E/S normalmente toma la forma de LED, pulsador, ambos o un parlante piezoeléctrico. La Figura B.2 muestra la conexión común para LED. Si está usando LEDs de 5V, no necesitará el resistor debido a que viene dentro del LED. Observe que el LED está polarizado; vea las especificaciones de los LEDs para identificar la distribución de pines. Con el LED conectado como se muestra, debe poner el pin del SX es estado bajo para encender el LED.

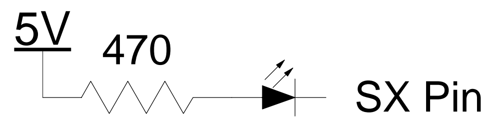

Figura B.2 Circuito para el LED

En la unidad 5, algunos ejercicios usan un pulsador y un piezoeléctrico para E/S (ver Figura B.3). El valor del resistor de 10K no es crítico. Cualquiera entre 10K y 22K (o incluso más) serviría. Si un proyecto necesita más interruptores, puede duplicar la parte del pulsador para otros pines. Simplemente use un resistor de pull up en el pin y conecte el botón a masa.

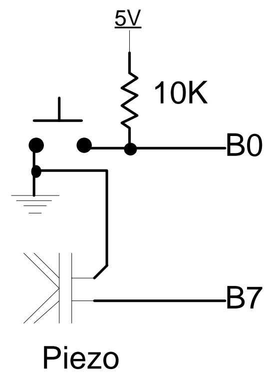

Figura B.3 Circuito con Parlante y Pulsador

No conecte un parlante común al pin del SX debido a que la carga puede dañar al chip SX. La mayoría de los circuitos integrados, incluyendo el SX, pueden manejar directamente un parlante piezoeléctrico.

Sobre la SX Demo Board (Plaqueta de Demostración del SX)

Si tiene una de las antiguas SX Demo Boards, toda la circuitería necesaria para estos ejercicios se encuentra presente en la plaqueta. En la Unidad 7, algunos de los programas usan una combinación de pulsador y LED, como la que se encuentra en la SX Demo Board (ver Figura B.4). Sin embargo, este circuito trabaja mejor cuando los resistores internos de pull up están activados en los pines del SX que están conectados a él.

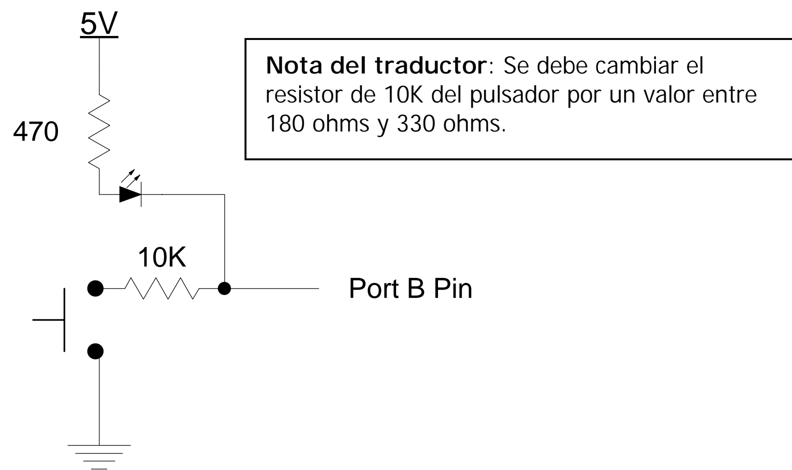

Figura B.4 – Combinación Pulsador/LED

El Proyecto Final

El proyecto final de este libro es un marcador telefónico por tonos. Con fines demostrativos, puede escuchar los tonos en un parlante piezoeléctrico (probablemente se escuchen bastante bajo, tal vez deba poner su oído sobre el parlante). Si quiere marcar un número realmente, necesitará dos cosas: un filtro y un amplificador.

Las notas de Scenix sobre el VP de generación de DTMF especifica los valores de los componentes para el filtro pasa bajos. Este filtro elimina la componente de alta frecuencia (un subproducto inevitable de la generación de tonos mediante PWM) para ingresar a líneas telefónicas. Conecte un resistor de 620 ohm al pin de salida del SX y un capacitor de .22uF desde el otro extremo del resistor a masa (las notas de Scenix piden 600 ohms y .2uF, pero estos valores son cercanos y fáciles de obtener). Esto hará que el tono se escuche más bajo aún. Se necesita amplificar estos tonos para poder introducirlos en la línea telefónica. Puede usar cualquier tipo de parlante amplificado, trazador de señal, o construir un pequeño amplificador usando un chip LM386 (ver Figura B.5) para conectar un parlante común de 8 ohms.

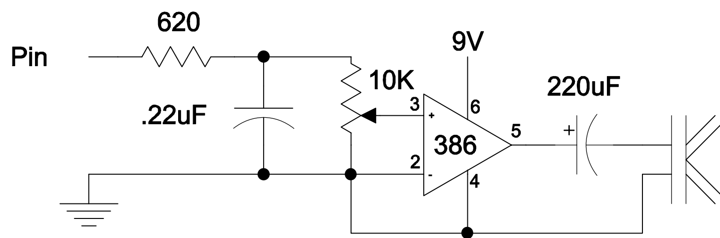

Figura B.5 – Amplificador Simple

Nota del Traductor: Se debe agregar un capacitor en serie entre la entrada (pin 3) del LM386 y el cursor del potenciómetro, para desacoplar la componente de continua que genera la PWM, con un valor entre 10nF y 100nF. También es recomendable, por la configuración del circuito, dejar sin conectar el pin 2 del LM386. La alimentación del LM386 se puede realizar con 5V.

Los programas y la información de este texto son para propósitos educativos, habiendo sido cuidadosamente probados, pero no están garantizados para ningún propósito en particular. El editor, el autor y el traductor no ofrecen ninguna garantía sobre la precisión, adecuación, o completitud de la información presentada, y no asumen ninguna responsabilidad por errores u omisiones en el texto. Tampoco asumen la responsabilidad por daños emergentes por el uso de la información de este texto, o por cualquier violación de los derechos intelectuales o de autor de terceros, que resulten del uso de esta información.

Versión en Castellano 1.0