Chapter 20LC

Lab C2 SiLabs 2: Input; Byte Operations

Contents

20LC Lab C2 SiLabs 2: Input; Byte Operations .......................... 1
  20LC.1 Port Pin Use in This Lab ............................................. 2
  20LC.2 Flip LED Blink Program, Recast with Subroutine .............. 2
  20LC.2.1 Some Oddities of PUSH AND POP ............................... 3
  20LC.2.2 Bit Input: “Blink LED if…” ........................................ 4
  20LC.3 Bit Input ..................................................................... 4
  20LC.3.1 Bit Input .................................................................. 4
  20LC.3.2 C language Equivalent: “If…” .................................... 7
  20LC.4 Byte Operations, In and Out ......................................... 7
  20LC.4.1 Hardware for Byte In, Byte Out .................................. 7
  20LC.4.2 Code for Byte In, Byte Out ........................................ 8
  20LC.5 Add Keypad Value to Running Sum ............................... 9
  20LC.5.1 Code for 8-bit Running Sum ...................................... 9
  20LC.5.2 16-bit Running Sum .................................................. 11
  20LC.6 You Can Disconnect Cables, etc. ................................... 14

REV 01; March 23, 2015.

1Revisions: check init caps in headings, space after §/3/15; add header and index (7/14); interchange Hi, Lo port ass’ms in 16-bit display, to be consistent with u4 port use; update display photos to show no-label display of 8 and 16-bit quantities (4/14); replace keypad hardware image to fit new display (3/14); add reference to 16-bit data display LCD (4/13); add note suggesting disconnect displays and keypad (11/11); correct pushbutton ref in text to P0.4 (3/11).
20LC.1 Port Pin Use in This Lab

Today, we make use of just one more PORT0 pin, in addition to the LED drive of lab C1: an input pin driven by a pushbutton.

Later in this lab, in § 20LC.4 on page 7, we also add byte input and output devices at two other ports, PORT1 (display) and PORT2 (keypad), and finally we also ask PORT0 to drive another 8-bit LCD display. That last addition does not conflict with the two PORT0 uses shown in fig. 1.

Figure 1: PORT0 pin use in today’s lab: a pushbutton input added

20LC.2 Bitflip LED Blink Program, Recast with Subroutine

Last time, we slowed the LED-blinking program by putting time-wasting DELAY code inline with the bitflip code. This works, but makes the code relatively hard to read, and clumsy to write. In a tiny program like Bitflip this hardly matters. But in a more complex program it is useful to break the code into modules, called subroutines, all of which are stitched together by a main calling program. Here, we will make that formal change to the delayed bitflip program of Lab C1. The behavior of the LED will be the same. The code will be easier read.

The code below is just the old code, rearranged—with a few details added that permit the main program to invoke the two subsidiary subroutines: one subroutine takes care of the SiLabs special initializations (we call this patch of code “USUAL_SETUP,” and it reappears in almost all of the programs in this series of labs). The other subroutine, DELAY, simply uses up about one second, to set the LED blink rate, as in Lab SiLabs µ1.

We have made one small change in “USUAL_SETUP,” relative to that used in the delay code of µ1: we have slowed the system clock by a factor 8, as the '410 permits, in order to save a little power. In this application, certainly this slowdown has no appreciable effect. It is not necessary, but we thought we’d show this option—which is the IDE’s default, but here imposed by the explicit command, GRL OSCICN, #04h ; sysclk = 24.5 Mhz /8.

We’ll list the program, and then look at some of its details.

```plaintext
; bitflip_delay_subroutine.a51 bitflip program, with delay for full-speed operation
$NOSYMBOLS ; keeps listing short
$INCLUDE (C:\MICRO\8051\RAISON\INC\c8051f410.inc)

ORG 0 ; tells assembler the address at which to place this code
ACALL USUAL_SETUP
SETB P0.0 ; start with LED OFF (it's active low)

;--- SUBROUTINES ----
FLIPIT: CPL P0.0 ; toggle LED
ACALL DELAY ; waste some time
SJMP FLIPIT ; do it forever

;--- SUBROUTINES ----
DELAY: PUSH ACC ; save the registers that this routine will mess up
PUSH B
PUSH 4 ; this saves register R4--in the zeroth set of registers
```
MOV A,#0 ; maximize two delay values (0 is max because dec before test)
MOV B,#0 ; ...and second loop delay value
MOV R4, #10h ; 1 second delay: this multiplies the 64K other loops

INNERLOOP: DJNZ B, INNERLOOP ; count down innermost loop, till inner hits zero
DJNZ ACC,INNERLOOP ; ...then dec second loop, and start inner again.
DJNZ R4, INNERLOOP ; now, with second at zero, decrement the outermost loop

POP 4
POP B
POP ACC
RET ; back to main program

USUAL_SETUP: ANL PCA0MD, #NOT(040h) ; Disable the WDT.
 ; Clear Watchdog Enable bit
ORL OSCICN, #04h ; sysclk = 24.5 Mhz / 8
 ; Enable the Port I/O Crossbar
MOV XBR1, #40h ; Enable Crossbar
RET
END

The subroutine is invoked by a CALL operation, here in the shorter ACALL version. (Subroutines and the stack are discussed in Classnotes µ2). Subroutine etiquette requires that the subroutine save any values it will mess up, so that the main program can operate properly on a return from the subroutine. The saving here is done with PUSH operations; the same registers are restored, before exit from the subroutine, with POP operations. RET pulls the “return” address off the stack, so that execution can resume at the main program’s instruction that follows the ACALL.

20LC.2.1 Some Oddities of PUSH AND POP

20LC.2.1.1 Potential Ambiguity in the Name “Rn”

One of the PUSHes is odd: PUSH 4 ; this saves register R4--in the zeroth set of registers. Why “PUSH 4” rather than “PUSH R4”? Because “R4,” surprisingly, is ambiguous: the 8051 has four sets of scratch registers, named R0...R7. The 8051 resolves the ambiguity by referring to two bits in its PSW (“Program Status Word”) register. After a reset, these two bits are at zero, so the controller uses the zeroth register set, by default.

But the broad-minded assembler program does not dare to presume which of the four register sets is intended. So, it requires that we describe the zeroth R4 by its on-chip address. “4” is the (unambiguous) address of the zeroth version of R4.2

20LC.2.1.2 “A” Wants to be Called “ACC”

As we noted in Classnotes µ2, PUSH and POP require that we call register A “ACC.” Don’t ask why!3

---

2We wish the assembler were bold enough to assume that the particular R4 intended is the one in use at the time of the reference. But this is not the way 8051 assemblers work. [ASK PAUL IF THERE’S AN OBJECTION TO THIS PROPOSAL]

3...because we don’t know.
The use of subroutines makes even this simple program easier for a reader to make sense of. The guts of the program sit near the start:

```assembly
FLIPIT: CPL P0.0 ; toggle LED
       ACALL DELAY ; waste some time
       SJMP FLIPIT ; do it forever
```

Appearing early, and all in one place, this code is easier to make sense of than the code of the delayed-bitflip program of Lab SiLabs 1, where the delay and initialization code was placed inline.

From this point, we will use subroutines regularly, in an effort to make our programs readable and also to permit building them up in stages. This effort recalls our methods in the analog part of this course: there, we tried to design and test subcircuits that we then could link as modules. The method eased our work in both design and analysis—and the motive for use of subroutines is the same.

### 20LC.3 Bit Input: “Blink LED if…”

In the first micro lab, we made an LED blink. As an application for an intelligent controller, that’s not very impressive. A ’555 oscillator can make an LED blink. Now we will modify the program just slightly, so that the LED blinks unless we press a pushbutton. That’s still not very impressive, you may protest: a pushbutton on the ’555 RESET line could achieve the same result.

Well, OK: fair enough. But we’d like to make a claim that the ability of the processor to do one thing under one condition, another thing under another condition amounts to a glimmering of intelligence. This ability is fundamental to what can make a computer seem smart. So, let’s add this capability to the blinker program.

### 20LC.3.1 Bit Input

#### 20LC.3.1.1 Hardware

When using buses—as you have seen in today’s class notes—responding to a bit input requires some hardware (a 3-state to get the information onto the data bus) and then a bit-testing operation on the accumulator register (also called “ACC” or just “A”).

Using a controller’s built-in port, as in the ’410 (which provides no buses), data inputs are much simpler. An input signal (either a bit as in bitflip.jf.a51, or a byte, as in § 20LC.4 on page 7) is tied directly to the controller’s port pin or pins: no need for a 3-state, because the pin accesses not a public bus but a private road into the controller.

The hardware to let us talk to the ’410 with a pushbutton thus is extremely simple: a pushbutton can ground the input signal while a pullup resistor otherwise pulls it to +5V.4 If switch bounce matters, then at least a capacitor must be added, to slow the edge.5 But let us omit that here, for maximum simplicity, since bounce does not matter in this program. Wire this on a breadboard strip.

---

4On some 8051/’410 ports even the pullup can be omitted. But at PORT 0, which we use here, no such pullup is present.
5As you know, a true debouncer for an SPST switch requires more than an RC circuit. It also requires a Schmitt trigger to square up the transition of the slow RC. But a processor input, unlike a true edge-triggered input such as a flip-flop clock, does not need that squaring up, as we will see next time, in Lab C3.
20LC.3.1.2 Code

Subroutines Tidy the Code Applying our new wisdom (new in § 20LC.2 on page 2) we again push the details of initializations and Delay out of the way of the main program loop, making them subroutines.

A Formal Novelty: Introducing Symbols to Make Code More Intelligible We have made some small changes to the bitflip code, intended to make the code more readable. We have used symbols—descriptive names—to stand in for the literal port pins used in this program.

Whereas in the original bitflip.a51 program we wrote “P0.0,” we here will replace the port-pin specification with a descriptive name, “BLUE_LED.” Similarly, we will replace the pin designation of the input pin, P0.4, with the description “PB” (“pushbutton).

So, whereas bitflip’s loop looked like...

\[ \text{FLIPIT: CPL P0.0 ; flip LED, ON, then OFF...} \]
\[ \text{SJMP FLIPIT} \]

...the loop in the flip_if.a51 program gives a reader a little help in divining what the loop does.

At the head of the program we define the two symbols—“BLUE_LED” and “PB” so that we can write these descriptive names rather than list the port pins. We use an assembler directive, EQU, to define symbolic names for the two pins:

\[ \text{BLUE_LED EQU P0.0} \]
\[ \text{PB EQU P0.4} \]

An assembler directive, like EQU, is a command addressed not to the 8051 but to the assembler. Each time the assembler encounters the symbol “BLUE_LED,” it will substitute “P0.0,” as it translates our assembly code into executable code. It will also do the equivalent substitution for “PB.”

Having defined these symbols we can make the flip-if code loop fairly intelligible. In § 20LC.3.1.3 we ask you to finish the program for us.

20LC.3.1.3 Your Task: Implement the “If...” in Assembly Language

Now we have set up the symbol “PB” to stand for the bit that is to be tested. We assign a label, “FLIPIT,” to the location to which we want the program to return when the pushbutton is pressed.

\[ \text{FLIPIT: ____ PB, FLIPIT ; hang up here, so long as pushbutton is pressed} \]
\[ \text{CPL BLUR_LED} \]

We would like you to fill in the 8051 instruction that will evoke this behavior. You may want to consult the one-page list of 8051 instructions that we have put together as a supplement to Class Micro 1. The same

---

6The use of symbols and labels is sometimes oversold with the description “self-documenting code.” We don’t claim that labels are quite that wonderful.

7This topic is treated in Classnotes µ2.

When you have filled in the code, the label and symbol offer strong clues to the logic of this loop. Perhaps you find FLIPIT: ____ PB, FLIPIT and CPL BLUE_LED cryptic still. But the symbols help, do they not?

**Program Flow Conditioned by Pushbutton**  
The novelty in the behavior of bitflip_if.a51—distinct from the novelty in the program’s listing—is the code’s testing of an input. The conditional jump command that we asked you to fill in implements the “if” in the “Blink...if...” This conditional (“if”...) keeps the program stuck on this line so long as the pushbutton holds P0.1 at a logic Low. Only when the pushbutton is released, permitting P0.1 to go High, does the program flow reach the next line, where the LED-flipping occurs.

Here is the full program listing:

```assembly
; bitflip_if.a51  toggle LED, slowly--if pushbutton NOT pressed
$NOSYMBOLS ; keeps listing short
$INCLUDE (C:\MICRO\8051\RAISON\INC\c8051f410.inc)
BLUE_LED EQU P0.0
PB EQU P0.4 ; pushbutton will determine whether LED is to toggle or not
ORG 0 ; tells assembler the address at which to place this code
SJMP STARTUP ; here code begins--with just a jump to start of 
; real program. ALL our programs will start thus
ORG 80h ; ...and here the program starts
STARTUP: ACALL USUAL_SETUP
CLR BLUE_LED ; start low, just to make it predictable
FLIPIT: ___ PB, FLIPIT ; hang up here, so long as pushbutton is pressed
    CPL BLUE_LED ; ...but flip the LED when button not pushed
    ACALL DELAY
    SJMP FLIPIT

;----SUBROUTINES -----
USUAL_SETUP: ; Disable the WDT.
    ANL PCA0MD, #NOT(040h) ; Clear Watchdog Enable bit
; Configure the Oscillator
    ORL OSCICN, #04h ; sysclk = 24.5 Mhz / 8, for lower power
; Enable the Port I/O Crossbar
    MOV XBR1, #40h ; Enable Crossbar
    RET

DELAY:  PUSH ACC ; save the registers that this routine will mess up
    PUSH B
    PUSH 4  ; this saves register R4--in the zeroth set of registers
    MOV A,#0 ; maximize two delay values (0 is max because dec before test)
    MOV B,#0 ; ...and second loop delay value
    MOV R4, #10h ; 1 second delay: this multiplies the 64K other loops
    INNERLOOP: DJNZ B, INNERLOOP ; count down innermost loop, till inner hits zero
    DJNZ ACC,INNERLOOP ; ...then dec second loop, and start inner again.
    DJNZ R4, INNERLOOP ; now, with second at zero, decrement the outermost loop
    POP 4
    POP B
    POP ACC
    RET ; back to main program

END
```

We found the blink rate a little low for our tastes. Try speeding it up by, say, a factor of four.
20LC.3.2 C language Equivalent: “If…”

C language code that would achieve the same result—flipping an LED unless a pushbutton grounds an input named “inbit”—looks a lot like the assembly language loop of § 20LC.3.1.2 on page 5. In C the “if” is explicit; in assembly language the “if” is expressed in the conditional jump operation that you filled in, in § 20LC.3.1.3 on page 5.

```c
if(!inbit) outbit = outbit;
else outbit = !outbit;
```

For code as simple as this, C offers no advantage over assembly.

20LC.4 Byte Operations, In and Out

We have used bit operations, so far, because they give such a quick reward for minimal wiring. Byte operations—with which we started on the Big Board branch of the micro labs—are just as simple to code, but require a very little more wiring effort. Let’s make that small effort, now, and do some byte operations.

20LC.4.1 Hardware for Byte In, Byte Out

20LC.4.1.1 Byte Input, from Keypad

As input device, let’s use the keypad, with its 16-pin DIP connector. On a breadboard strip you can convert the DIP’s 8 data lines—which unfortunately lie on two sides of the connector—into the 8-in-a-row form that will be convenient for wiring to the ’410:

![Figure 3: DIP connector from keypad, adapted to in-line flat cable](image)

From this in-line set of 8 data lines, run a flat ribbon cable to the ’410’s PORT 2.

![Figure 4: Keypad cable wired to ’410: P2.7 split from cable](image)
The D7 line on the cable, you’ll notice, comes not from the ’410’s P2.7 but from the far end of the C2-neutralizing resistive network described in Lab SiLabs µ 1.

20LC.4.1.2 Keypad In, Byte Display Out

We’ll first do a byte-in, byte-out transfer, keypad to display. As a way to see the byte-wide output from the ’410, let’s use the LCD display board. It provides 8 bits for data, 16 bits for what, on the Dallas branch is address. For the one-byte display of the current program we will use the low byte (D7…D0)of the data display—one of the two rightmost byte-wide connectors on the LCD board.

![Figure 5: Hardware connections for byte-in, byte-out keypad to display](image)

Though PORT0 here drives the display, you need not disconnect the LED at P0.0 or the pushbutton at P0.4 unless you choose to. Neither should interfere with the ’410’s drive of the display: each applies only a pullup resistor to its pin (unless you are so rash as to press the pushbutton while using PORT0 to drive the LCD display!).

20LC.4.2 Code for Byte In, Byte Out

The code to transfer a byte from keypad to display is as simple as you would expect it to be:

```
TRANSFER: MOV DISPLAY, KEYPAD
SJMP TRANSFER
```

20LC.4.2.1 A Task: Define the Symbols

Again we have (mischievously?) not quite finished the code below. You will need to use the EQU assembler directive to define the two symbols used in the program: DISPLAY and KEYPAD. The form will be—as the incomplete listing below indicates—
When you’ve satisfied yourself that the program can, indeed, transfer a byte from keypad to display, we’ll ask the ’410 to do something a shade more exciting.

20LC.5 Add Keypad Value to Running Sum

Perhaps it pains you to see your smart little ’410 simply transferring data. OK. Let’s allow it to show that it can also add.

20LC.5.1 Code for 8-bit Running Sum

This program adds the keypad value to a running sum, and outputs that sum to the display. We will do this first with an 8-bit output, then 16-bit (just to show that the 8-bit controller is not necessarily restricted to 8-bit operations). When the running sum overflows, the 8-bit output remains valid “modulo 8,” in the jargon. The overflow information is lost.

Here is the core loop of the program:
Transfer: MOV A, RUN_SUM ; recall running sum
ADD A, KEYPAD ; form new sum. Note that result lands in A
MOV RUN_SUM, A ; ...save it
MOV DISPLAY, A ; ...and show it
ACALL DELAY
SJMP Transfer ; ...forever

To make sense of this code, you need to recall that the result of ADD goes to the accumulator, the A register. So, each pass through the loop updates both the running sum (RUN_SUM) and the display.

As you try this program, you might start by setting up 01h on the keypad. The summing then amounts just to continually incrementing, of course. When you're feeling more adventurous, try keypad value FFh.

; keysum_8bit.a51 shows sum of keypad and running sum; decimal adjust after initial binary display
; $NOSYMBOLS ; keeps listing short
$INCLUDE (C:\MICRO\8051\RAISON\INC\c8051f410.inc)
STACKBOT EQU 80h ; put stack at start of scratch indirectly-addressable block (80h and up)
DISPLAY EQU P1 ; so-called Data byte on LCD board
KEYPAD EQU P2
RUN_SUM EQU R7 ; this choice is arbitrary
DELAY_MULTIPLIER EQU 08h ; about half-second delay, at div-by-8 clock rate
ORG 0h
LMP STARTUP
ORG 018h
STARTUP: MOV SP, #STACKBOT-1
ACALL USUAL_SETUP
; Initialize running sum
CLR A
MOV RUN_SUM, A ; clear running sum
Transfer: MOV A, RUN_SUM ; recall running sum
ADD A, KEYPAD ; form new sum
; ADD A, #0 ; For decimal sum--try after watching binary addition
MOV RUN_SUM, A ; ...save it
MOV DISPLAY, A ; ...and show it
ACALL DELAY
SJMP Transfer ; ...forever

;----SUBROUTINES ----
DELAY: PUSH ACC ; save the registers that this routine will mess up
PUSH B
PUSH 4 ; this saves register R4--in the zeroth set of registers
MOV A,#0 ; maximize two delay values (0 is max because dec before test)
MOV B,#0 ; ...and second loop delay value
MOV R4, #DELAY_MULTIPLIER ; a more general way to specify delayu: this multiplies the 64K other loops
INNERLOOP: DJNZ B, INNERLOOP ; count down innermost loop, till inner hits zero
DJNZ ACC,INNERLOOP ; ...then dec second loop, and start inner again.
DJNZ R4, INNERLOOP ; now, with second at zero, decrement the outermost loop
POP 4 ; restore saved registers
POP B
POP ACC
RET ; back to main program

USUAL_SETUP: ; Disable the WDT.
ANL PCA0MD, #NOT(040h) ; Clear Watchdog Enable bit
; Configure the Oscillator;
ORL OSCICN, #04h ; sysclk = 24.5 Mhz / 8, for lower power
; Enable the Port I/O Crossbar
MOV XBR1, #40h ; Enable Crossbar
RET
END
20LC.5.1.1 Delay Routine Marginally Amended

We made one small change to the Delay routine, in § 20LC.5.1 on page 9: rather than fix the duration of Delay within that subroutine, as we did earlier (for example, in § 20LC.2 on page 2), we set the value of the delay multiplier at the head of the program. The multiplier, in this routine, determines how many times the 16-bit countdown loop will be repeated (each loop taking about 80ms).

This change makes Delay more versatile, and also illustrates a point concerning the design of programs: it’s a good idea to initialize constants up at the start of a program, where they are easy to see and to change. If you decide to change the Delay duration from 1 second to two, you don’t want to have to search the program listing so as to locate the place where that duration is set.

20LC.5.1.2 8-bit Output Amended to Decimal Form

Binary addition makes efficient use of the running-sum and keypad bytes, permitting use of all 256 combinations. But when a display is intended for humans—with their many fingers, and their peculiar decimal counting system—a decimal output often is preferable. The 8051 knows how to restrict its output to the decimal values, so long as we give it keypad values that also are limited to decimal values.

All we need do, in order to see this behavior, is to un-comment the instruction “DA” in the program of § 20LC.5.1 on page 9. DA (“Decimal Adjust”) operates on the Accumulator (A register), noting when a “half-carry” from the low nybble occurs. So, for example, if A holds the value 09h and we add one, in binary the next value would be 0Ah. But DA notes the “half-carry” and amends the accumulator value appropriately: the “half-carry” tells the processor to roll over the low nybble to 0 while incrementing the high nybble. The result thus becomes 10h, the decimal count, indeed, that must succeed 09h.

We should note what DA cannot do: it cannot transform a hexadecimal value to decimal; it works properly only after an operation, like ADD, that affects flags appropriately—particularly, the half-carry flag. The details of flag effects appear in the descriptions of each processor operation, in the 8051 instruction set reference.

The first time you do an operation that depends on flag behaviors, you would be wise to check this behavior in the instruction set reference. INC (increment) and DEC affect no flags, for example. That is a behavior one could not anticipate from any principle that we can perceive.

It is, of course, possible to do a generalized conversion from a binary value to decimal. But this process calls for a multi-step algorithm, and is neither simple nor quick. DA neatly covers the present task.

20LC.5.2 16-bit Running Sum

Now let’s allow the 8051 show that it can handle values larger than one byte. We don’t often ask it to do so—and, in fact, are inclined to keep its computation tasks minimal. But at a price of speed, an 8-bit processor can concatenate operations so as to operate on values larger than what fits in a single byte.

Here, we will keep the running sum to 16 bits rather than 8, and we will display it on the LCD board using four hex digits rather than two, as for the byte sum. We’ll exercise this hardware with an amended running-sum program.
20LC.5.2.1 Hardware for 16-bit Display

20LC.5.2.2 Two-Byte Output, to Display

On the LCD board you will find two 8-bit ‘data’ connectors, labelled “D15...D8” and “D7...D0.” These will accept your micro’s 16-bit output. Set the right-hand DIP switch to “16” rather than “8mux.” This makes the two 8-bit ports independent. (“8mux,” in contrast, would merge the inputs, so that they could be separated only by the separate assertions of the two latch ENABLE*s. That is an option you will not use, here.)

We will use three byte-wide ports all at once. On the left, in fig. 6 is a reminder of how they can be wired to the ’410, and on the right keypad goes in as before, while two bytes go from controller to LCD.

![Image of LCD board with 8-bit data connectors](image)

Figure 6: All three ’410 ports can be wired with flat cables

The keypad already is wired to P2, and we will leave it there. Low-byte of data is done, too, wired in § 20LC.5.2.2. You now only add a flat cable between micro’s P1 and the LCD’s “data high” connector (D0...D15).

LCD switch settings:

- probably you will want to suppress labels, using the LABELS slide switch at the bottom right of the LCD board;
- you’ll want to suppress the upper display line, which shows the 16 inputs labelled “address” on the LCD board. Suppress this line by setting the LINES switch to “1.”
- In order to show 16 bits of data, you need to set the DATA switch to “16.”

20LC.5.2.3 Code for 16-bit Display

This version of the keypad-summing program forms a 16-bit running sum by taking advantage of the Carry flag (CY) which warns us when an 8-bit sum has overflowed.

The oddity of the way the 16-bit sum is formed is explained in Classnotes µ 2 (last section), but we’ll reiterate the point here, in summary form:
MOV A, RUN_SUM_HI ; Get hi byte
ADDC A, #0 ; If a carry from low sum, incorporate it

Adding zero certainly looks pointless, at first glance. It is not pointless, though, when used with ADDC, the form of addition that incorporates the CY flag as an input. If the prior ADD (forming the low byte of the sum) has generated a Carry, ADDC will increment the high byte. Otherwise, the high byte remains as it was. This is just the behavior we require.

; keysum_16bit.a51 shows 16-bit sum of keypad and running sum;

$NOSYMBOLS ; keeps listing short, lest...
$INCLUDE (C:\MICRO\8051\RAISON\INC\c8051f410.inc) ; ...this line might produce huge list
; of symbol definitions (all ‘51 registers)
$INCLUDE (C:\MICRO\8051\RAISON\INC\VECTORS320.INC) ; Tom’s vectors definition file
STACKBOT EQU 80h ; put stack at start of scratch indirectly-addressable block (80h and up)

DISPLAY_HI EQU P1 ; high byte of LCD address
DISPLAY_LO EQU P0 ; low byte of LCD address
KEYPAD EQU P2
DELAY_MULTIPLIER EQU 06h ; half-second delay
; multiplier value stored in R4
ORG 0h
LJMP STARTUP

ORG 080h
STARTUP: MOV SP, #STACKBOT-1
ACALL USUAL_SETUP

; Initialize running sum (zero it)
MOV RUN_SUM_HI, #0 ; clear running sum
MOV RUN_SUM_LO, #0

Transfer: MOV A, RUN_SUM_LO ; recall running sum (lo byte)
ADD A, KEYPAD ; form new sum (low byte)
;
DA A
MOV RUN_SUM_LO, A ; ...save it
MOV DISPLAY_LO, A ; ...and show it
MOV A, RUN_SUM_HI ; Get hi byte
ADDC A, #0 ; If a carry from low sum, incorporate it
;
DA A
MOV DISPLAY_HI, A ; ...and show high byte of 16-bit sum
MOV RUN_SUM_HI, A ; ...and save it
ACALL DELAY
SJMP Transfer ; ...forever

;----SUBROUTINES ----

DELAY: PUSH ACC ; save the registers that this routine will mess up
PUSH B
PUSH 4 ; this saves register R4--in the zeroth set of registers
MOV A,#0 ; maximize two delay values (0 is max because dec before test)
MOV B,#0 ; ...and second loop delay value
MOV R4, #DELAY_MULTIPLIER ; a more general way to specify delay: this multiplies the 64K other loops

INNERLOOP: DJNZ B, INNERLOOP ; count down innermost loop, till inner hits zero
DJNZ ACC, INNERLOOP ; ...then dec second loop, and start inner again.
DJNZ R4, INNERLOOP ; now, with second at zero, decrement the outermost loop
POP 4 ; restore saved registers
POP B
POP ACC
RET ; back to main program

;------------------------------------------------------------------------------
Try the keypad value FFh again, as you did in the 8-bit summing (§ 20LC.5.1 on page 9). With the earlier 8-bit display, FFh produced a decrementing value. Now, the 16-bit version makes the magic of “FFh as minus one” somewhat less magical: the carries into the high byte now are visible. The lower byte does, indeed, decrement; but at the same time the high byte increments. We stand backstage now, and we can see what the byte-addition magician was doing all along.

Again, we have planted decimal-adjust (DA) instructions, which you may use if you like, by removing the comment semicolons. Once again, DA works properly only so long as the keypad input presents values in the decimal range (no fair using A, B…, etc.).

20LC.6 … You Can Disconnect Cables, etc.

You will not need today’s displays and keypad and associated cables in the next lab. You may disconnect these.

You can leave in place the pushbutton used to disable the LED’s blinking in § 2 on page 4. You will not need it again for that purpose, and you ought not to press it randomly (some students were startled to find that doing so disabled some later programs; we were startled to find that they were startled). The pushbutton will see use again when you try interrupts in Lab C4.
Index

ACC
  vs “A” SiLabs (lab), 3
ADDC (lab), 12

CALL
  SiLabs (lab), 2

DA (lab), 11

EQU (lab), 8

IF
  bit test equivalents (lab), 7

POP
  SiLabs (lab), 3

PUSH
  SiLabs (lab), 3

subroutine
  SiLabs (lab), 2