data from the buffer at its leisure without losing any data from the port. Likewise, a typical interrupt-driven output system (that gets an interrupt whenever the output device is ready to accept more data) can remove data from a buffer whenever the peripheral device is ready to accept new data.

3.6 Laboratory Exercises

In this laboratory you will use the "SIMX86.EXE" program found in the Chapter Three subdirectory. This program contains a built-in assembler (compiler), debugger, and interrupter for the x86 hypothetical CPUs. You will learn how to write basic x86 assembly language programs, assemble (compile) them, modify the contents of memory, and execute your x86 programs. You will also experiment with memory-mapped I/O, I/O-mapped input/output, DMA, and polled as well as interrupt-driven I/O systems.

In this set of laboratory exercises you will use the SIMx86.EXE program to enter, edit, initialize, and emulate x86 programs. This program requires that you install two files in your WINDOWS\SYSTEM directory. Please see the README.TXT file in the CH3 subdirectory for more details.

3.6.1 The SIMx86 Program – Some Simple x86 Programs

To run the SIMx86 program double click on its icon or choose run from the Windows file menu and enter the pathname for SIMx86. The SIMx86 program consists of three main screen that you can select by clicking on the *Editor, Memory, or Emulator* notebook tabs in the window. By default, SIMx86 opens the Editor screen. From the Editor screen you can edit and assemble x86 programs; from Memory screen you can view and modify the contents of memory; from the Emulator screen you execute x86 programs and view x86 programs in memory.

The SIMx86 program contains two menu items: File and Edit. These are standard Windows menus so there is little need to describe their operation except for two points. First, the New, Open, Save, and Save As items under the file menu manipulate the data in the text editor box on the Editor screen, they do not affect anything on the other screens. Second, the Print menu item in the File menu prints the source code appearing in the text editor if the Editor screen is active, it prints the entire form if the Memory or Emulator screens are active.

To see how the SIMx86 program operates, switch to the Editor screen (if you are not already there). Select "Open" from the File menu and choose "EX1.X86" from the Chapter Three subdirectory. That file should look like the following:

```
mov     ax, [1000]
mov     bx, [1002]
add     ax, bx
sub     ax, 1
mov     bx, ax
add     bx, ax
add     ax, bx
halt
```

This short code sequence adds the two values at location 1000 and 1002, subtracts one from their sum, and multiplies the result by three ((ax + ax) + ax) = ax*3), leaving the result in ax and then it halts.

On the Editor screen you will see three objects: the editor window itself, a box that holds the "Starting Address," and an "Assemble" button. The "Starting Address" box holds a hexadecimal number that specifies where the assembler will store the machine code for the x86 program you write with the editor. By default, this address is zero. About the only time you should change this is when writing interrupt service routines since the default reset address is zero. The "Assemble" button directs the SIMx86 program to con-

vert your assembly language source code into x86 machine code and store the result beginning at the Starting Address in memory. Go ahead and press the "Assemble" button at this time to assemble this program to memory.

Now press the "Memory" tab to select the memory screen. On this screen you will see a set of 64 boxes arranged as eight rows of eight boxes. To the left of these eight rows you will see a set of eight (hexadecimal) memory addresses (by default, these are 0000, 0008, 0010, 0018, 0020, 0028, 0030, and 0038). This tells you that the first eight boxes at the top of the screen correspond to memory locations 0, 1, 2, 3, 4, 5, 6, and 7; the second row of eight boxes correspond to locations 8, 9, A, B, C, D, E, and F; and so on. At this point you should be able to see the machine codes for the program you just assembled in memory locations 0000 through 000D. The rest of memory will contain zeros.

The memory screen lets you look at and possibly modify 64 bytes of the total 64K memory provided for the x86 processors. If you want to look at some memory locations other than 0000 through 003F, all you need do is edit the first address (the one that currently contains zero). At this time you should change the starting address of the memory display to 1000 so you can modify the values at addresses 1000 and 1002 (remember, the program adds these two values together). Type the following values into the corresponding cells: at address 1000 enter the value 34, at location 1001 the value 12, at location 1002 the value 01, and at location 1003 the value 02. Note that if you type an illegal hexadecimal value, the system will turn that cell red and beep at you.

By typing an address in the memory display starting address cell, you can look at or modify locations almost anywhere in memory. Note that if you enter a hexadecimal address that is not an even multiple of eight, the SIMx86 program disable up to seven cells on the first row. For example, if you enter the starting address 1002, SIMx86 will disable the first two cells since they correspond to addresses 1000 and 1001. The first active cell is 1002. Note the SIMx86 reserves memory locations FFF0 through FFFF for memory-mapped I/O. Therefore, it will not allow you to edit these locations. Addresses FFF0 through FFF7 correspond to read-only input ports (and you will be able to see the input values even though SIMx86 disables these cells). Locations FFF8 through FFFF are write-only output ports, SIMx86 displays garbage values if you look at these locations.

On the Memory page along with the memory value display/edit cells, there are two other entry cells and a button. The "Clear Memory" button clears memory by writing zeros throughout. Since your program's object code and initial values are currently in memory, you should not press this button. If you do, you will need to reassemble your code and reenter the values for locations 1000 through 1003.

The other two items on the Memory screen let you set the interrupt vector address and the reset vector address. By default, the reset vector address contains zero. This means that the SIMx86 begins program execution at address zero whenever you reset the emulator. Since your program is currently sitting at location zero in memory, you should not change the default reset address.

The "Interrupt Vector" value is FFFF by default. FFFF is a special value that tells SIMx86 "there is no interrupt service routine present in the system, so ignore all interrupts." Any other value must be the address of an ISR that SIMx86 will call whenever an interrupt occurs. Since the program you assembled does not have an interrupt service routine, you should leave the interrupt vector cell containing the value FFFF.

Finally, press the "Emulator" tab to look at the emulator screen. This screen is much busier than the other two. In the upper left hand corner of the screen is a data entry box with the label IP. This box holds the current value of the x86 *instruction pointer* register. Whenever SIMx86 runs a program, it begins execution with the instruction at this address. Whenever you press the reset button (or enter SIMx86 for the first time), the IP register contains the value found in the reset vector. If this register does not contain zero at this point, press the reset button on the Emulator screen to reset the system.

Immediately below the ip value, the Emulator page *disassembles* the instruction found at the address in the ip register. This is the very next instruction that SIMx86 will execute when you press the "Run" or "Step" buttons. Note that SIMx86 does not obtain this

instruction from the source code window on the Editor screen. Instead, it decodes the opcode in memory (at the address found in ip) and generates this string itself. Therefore, there may be minor differences between the instruction you wrote and the instruction SIMx86 displays on this page. Note that a disassembled instruction contains several numeric values in front of the actual instruction. The first (four-digit) value is the memory address of that instruction. The next pair of digits (or the next three pairs of digits) are the opcodes and possible instruction operand values. For example, the mov ax, [1000] instruction's machine code is C6 00 10 since these are the three sets of digits appearing at this point.

Below the current disassembled instruction, SIMx86 displays 15 instructions it disassembles. The starting address for this disassemble is *not* the value in the ip register. Instead, the value in the lower right hand corner of the screen specifies the starting disassembly address. The two little arrows next to the disassembly starting address let you quickly increment or decrement the disassembly starting address. Assuming the starting address is zero (change it to zero if it is not), press the down arrow. Note that this increments the starting address by one. Now look back at the disassembled listing. As you can see, pressing the down arrow has produced an interesting result. The first instruction (at address 0001) is "****". The four asterisks indicate that this particular opcode is an illegal instruction opcode. The second instruction, at address 0002, is not ax. Since the program you assembled did not contain an illegal opcode or a not ax instruction, you may be wondering where these instructions came from. However, note the starting address of the first instruction: 0001. This is the second byte of the first instruction in your program. In fact, the illegal instruction (opcode=00) and the not ax instruction (opcode=10) are actually a disassembly of the mov ax, [1000] two-byte operand. This should clearly demonstrate a problem with disassembly – it is possible to get "out of phase" by specify a starting address that is in the middle of a multi-byte instruction. You will need to consider this when disassembling code.

In the middle of the Emulator screen there are several buttons: Run, Step, Halt, Interrupt, and Reset (the "Running" box is an annunciator, not a button). Pressing the Run button will cause the SIMx86 program to run the program (starting at the address in the ip register) at "full" speed. Pressing the Step button instructs SIMx86 to execute only the instruction that ip points at and then stop. The Halt button, which is only active while a program is running, will stop execution. Pressing the Interrupt button generates an interrupt and pressing the Reset button resets the system (and halts execution if a program is currently running). Note that pressing the Reset button clears the x86 registers to zero and loads the ip register with the value in the reset vector.

The "Running" annunciator is gray if SIMx86 is not currently running a program. It turns red when a program is actually running. You can use this annunciator as an easy way to tell if a program is running if the program is busy computing something (or is in an infinite loop) and there is no I/O to indicate program execution.

The boxes with the ax, bx, cx, and dx labels let you modify the values of these registers while a program is not running (the entry cells are not enabled while a program is actually running). These cells also display the current values of the registers whenever a program stops or between instructions when you are stepping through a program. Note that while a program is running the values in these cells are static and do not reflect their current values.

The "Less" and "Equal" check boxes denote the values of the less than and equal flags. The x86 cmp instruction sets these flags depending on the result of the comparison. You can view these values while the program is not running. You can also initialize them to true or false by clicking on the appropriate box with the mouse (while the program is not running).

In the middle section of the Emulator screen there are four "LEDs" and four "toggle switches." Above each of these objects is a hexadecimal address denoting their memory-mapped I/O addresses. Writing a zero to a corresponding LED address turns that LED "off" (turns it white). Writing a one to a corresponding LED address turns that LED

"on" (turns it red). Note that the LEDs only respond to bit zero of their port addresses. These output devices ignore all other bits in the value written to these addresses.

The toggle switches provide four memory-mapped input devices. If you read the address above each switch SIMx86 will return a zero if the switch is off. SIMx86 will return a one if the switch is in the on position. You can toggle a switch by clicking on it with the mouse. Note that a little rectangle on the switch turns red if the switch is in the "on" position.

The two columns on the right side of the Emulate screen ("Input" and "Output") display input values read with the get instruction and output values the put instruction prints.

For this first exercise, you will use the Step button to single step through each of the instructions in the EX1.x86 program. First, begin by pressing the Reset button[22]. Next, press the Step button once. Note that the values in the ip and ax registers change. The ip register value changes to 0003 since that is the address of the next instruction in memory, ax's value changed to 1234 since that's the value you placed at location 1000 when operating on the Memory screen. Single step through the remaining instructions (by repeatedly pressing Step) until you get the "Halt Encountered" dialog box.

For your lab report: explain the results obtained after the execution of each instruction. Note that single-stepping through a program as you've done here is an excellent way to ensure that you fully understand how the program operates. As a general rule, you should always single-step through every program you write when testing it.

3.6.2 Simple I/O-Mapped Input/Output Operations

Go to the Editor screen and load the EX2.x86 file into the editor. This program introduces some new concepts, so take a moment to study this code:

```
                mov     bx, 1000
a:              get
                mov     [bx], ax
                add     bx, 2
                cmp     ax, 0
                jne     a

                mov     cx, bx
                mov     bx, 1000
                mov     ax, 0
b:              add     ax, [bx]
                add     bx, 2
                cmp     bx, cx
                jb      b

                put
                halt
```

The first thing to note are the two strings "a:" and "b:" appearing in column one of the listing. The SIMx86 assembler lets you specify up to 26 statement *labels* by specifying a single alphabetic character followed by a colon. Labels are generally the operand of a jump instruction of some sort. Therefore, the "jne a" instruction above really says "jump if not equal to the statement prefaced with the 'a:' label" rather than saying "jump if not equal to location ten (0Ah) in memory."

Using labels is much more convenient than figuring out the address of a target instruction manually, especially if the target instruction appears later in the code. The SIMx86 assembler computes the address of these labels and substitutes the correct address

22. It is a good idea to get in the habit of pressing the Reset button before running or stepping through any program.

for the operands of the jump instructions. Note that you *can* specify a numeric address in the operand field of a jump instruction. However, all numeric addresses must begin with a decimal digit (even though they are hexadecimal values). If your target address would normally begin with a value in the range A through F, simply prepend a zero to the number. For example, if "jne a" was supposed to mean "jump if not equal to location 0Ah" you would write the instruction as "jne 0a".

This program contains two loops. In the first loop, the program reads a sequence of values from the user until the user enters the value zero. This loop stores each word into successive memory locations starting at address 1000h. Remember, each word read by the user requires two bytes; this is why the loop adds two to bx on each iteration.

The second loop in this program scans through the input values and computes their sum. At the end of the loop, the code prints the sum to the output window using the put instruction.

For your lab report: single-step through this program and describe how each instruction works. Reset the x86 and run this program at full speed. Enter several values and describe the result. Discuss the get and put instruction. Describe why they do I/O-mapped input/output operations rather than memory-mapped input/output operations.

3.6.3 Memory Mapped I/O

From the Editor screen, load the EX3.x86 program file. That program takes the following form (the comments were added here to make the operation of this program clearer):

```
a:              mov     ax, [fff0]
                mov     bx, [fff2]

                mov     cx, ax      ;Computes Sw0 and Sw1
                and     cx, bx
                mov     [fff8], cx

                mov     cx, ax      ;Computes Sw0 or Sw1
                or      cx, bx
                mov     [fffa], cx

                mov     cx, ax      ;Computes Sw0 xor Sw1
                mov     dx, bx      ;Remember, xor = AB' + A'B
                not     cx
                not     bx
                and     cx, bx
                and     dx, ax
                or      cx, dx
                mov     [fffc], cx

                not     cx          ;Computes Sw0 = Sw1
                mov     [fffe], cx  ;Remember, equals = not xor

                mov     ax, [fff4]  ;Read the third switch.
                cmp     ax, 0       ;See if it's on.
                je      a           ;Repeat this program while off.
                halt
```

Locations 0FFF0h, 0FFF2h, and 0FFF4h correspond to the first three toggle switches on the Execution page. These are memory-mapped I/O devices that put a zero or one into the corresponding memory locations depending upon whether the toggle switch is in the on or off state. Locations 0FFF8h, 0FFFAh, 0FFFCh, and 0FFFEh correspond to the four LEDs. Writing a zero to these locations turns the corresponding LED off, writing a one turns it on.

This program computes the logical and, or, xor, and xnor (not xor) functions for the values read from the first two toggle switches. This program displays the results of these functions on the four output LEDs. This program reads the value of the third toggle switch to determine when to quit. When the third toggle switch is in the on position, the program will stop.

For your lab report: run this program and cycle through the four possible combinations of on and off for the first two switches. Include the results in your lab report.

3.6.4 DMA Exercises

In this exercise you will start a program running (EX4.x86) that examines and operates on values found in memory. Then you will switch to the Memory screen and modify values in memory (that is, you will directly access memory while the program continues to run), thus simulating a peripheral device that uses DMA.

The EX4.x86 program begins by setting memory location 1000h to zero. Then it loops until one of two conditions is met – either the user toggles the FFF0 switch or the user changes the value in memory location 1000h. Toggling the FFF0 switch terminates the program. Changing the value in memory location 1000h transfers control to a section of the program that adds together *n* words, where *n* is the new value in memory location 1000h. The program sums the words appearing in contiguous memory locations starting at address 1002h. The actual program looks like the following:

```
d:      mov       cx, 0                 ;Clear location 1000h before we
        mov       [1000], cx            ; begin testing it.

; The following loop checks to see if memory location 1000h changes or if
; the FFF0 switch is in the on position.

a:      mov       cx, [1000]            ;Check to see if location 1000h
        cmp       cx, 0                 ; changes. Jump to the section that
        jne       c                     ; sums the values if it does.

        mov       ax, [fff0]            ;If location 1000h still contains zero,
        cmp       ax, 0                 ; read the FFF0 switch and see if it is
        je        a                     ; off. If so, loop back. If the switch
        halt                            ; is on, quit the program.

; The following code sums up the "cx" contiguous words of memory starting at
; memory location 1002. After it sums up these values, it prints their sum.

c:      mov       bx, 1002              ;Initialize BX to point at data array.
        mov       ax, 0                 ;Initialize the sum
b:      add       ax, [bx]              ;Sum in the next array value.
        add       bx, 2                 ;Point BX at the next item in the array.
        sub       cx, 1                 ;Decrement the element count.
        cmp       cx, 0                 ;Test to see if we've added up all the
        jne       b                     ; values in the array.

        put                             ;Print the sum and start over.
        jmp       d
```

Load this program into SIMx86 and assemble it. Switch to the Emulate screen, press the Reset button, make sure the FFF0 switch is in the off position, and then run the program. Once the program is running switch to the memory screen by pressing the Memory tab. Change the starting display address to 1000. Change the value at location 1000h to 5. Switch back to the emulator screen. Assuming memory locations 1002 through 100B all contain zero, the program should display a zero in the output column.

Switch back to the memory page. What does location 1000h now contain? Change the L.O. bytes of the words at address 1002, 1004, and 1006 to 1, 2, and 3, respectively. Change

the value in location 1000h to three. Switch to the Emulator page. Describe the output in your lab report. Try entering other values into memory. Toggle the FFF0 switch when you want to quit running this program.

For your lab report: explain how this program uses DMA to provide program input. Run several tests with different values in location 1000h and different values in the data array starting at location 1002. Include the results in your report.

For additional credit: Store the value 12 into memory location 1000. Explain why the program prints *two* values instead of just one value.

3.6.5 Interrupt Driven I/O Exercises

In this exercise you will load *two* programs into memory: a main program and an interrupt service routine. This exercise demonstrates the use of interrupts and an interrupt service routine.

The main program (EX5a.x86) will constantly compare memory locations 1000h and 1002h. If they are not equal, the main program will print the value of location 1000h and then copy this value to location 1002h and repeat this process. The main program repeats this loop until the user toggles switch FFF0 to the on position. The code for the main program is the following:

```
a:      mov       ax, [1000]            ;Fetch the data at location 1000h and
        cmp       ax, [1002]            ; see if it is the same as location
        je        b                     ; 1002h. If so, check the FFF0 switch.
        put                             ;If the two values are different, print
        mov       [1002], ax            ; 1000h's value and make them the same.

b:      mov       ax, [fff0]            ;Test the FFF0 switch to see if we
        cmp       ax, 0                 ; should quit this program.
        je        a
        halt
```

The interrupt service routine (EX5b.x86) sits at location 100h in memory. Whenever an interrupt occurs, this ISR simply increments the value at location 1000h by loading this value into ax, adding one to the value in ax, and then storing this value back to location 1000h. After these instructions, the ISR returns to the main program. The interrupt service routine contains the following code:

```
        mov       ax, [1000]            ;Increment location 1000h by one and
        add       ax, 1                 ; return to the interrupted code.
        mov       [1000], ax
        iret
```

You must load and assemble both files before attempting to run the main program. Begin by loading the main program (EX5a.x86) into memory and assemble it at address zero. Then load the ISR (EX5b.x86) into memory, set the Starting Address to 100, and then assemble your code. **Warning:** *if you forget to change the starting address you will wipe out your main program when you assemble the ISR. If this happens, you will need to repeat this procedure from the beginning.*

After assembling the code, the next step is to set the interrupt vector so that it contains the address of the ISR. To do this, switch to the Memory screen. The interrupt vector cell should currently contain 0FFFFh (this value indicates that interrupts are disabled). Change this to 100 so that it contains the address of the interrupt service routine. This also enables the interrupt system.

Finally, switch to the Emulator screen, make sure the FFF0 toggle switch is in the off position, reset the program, and start it running. Normally, nothing will happen. Now press the interrupt button and observe the results.

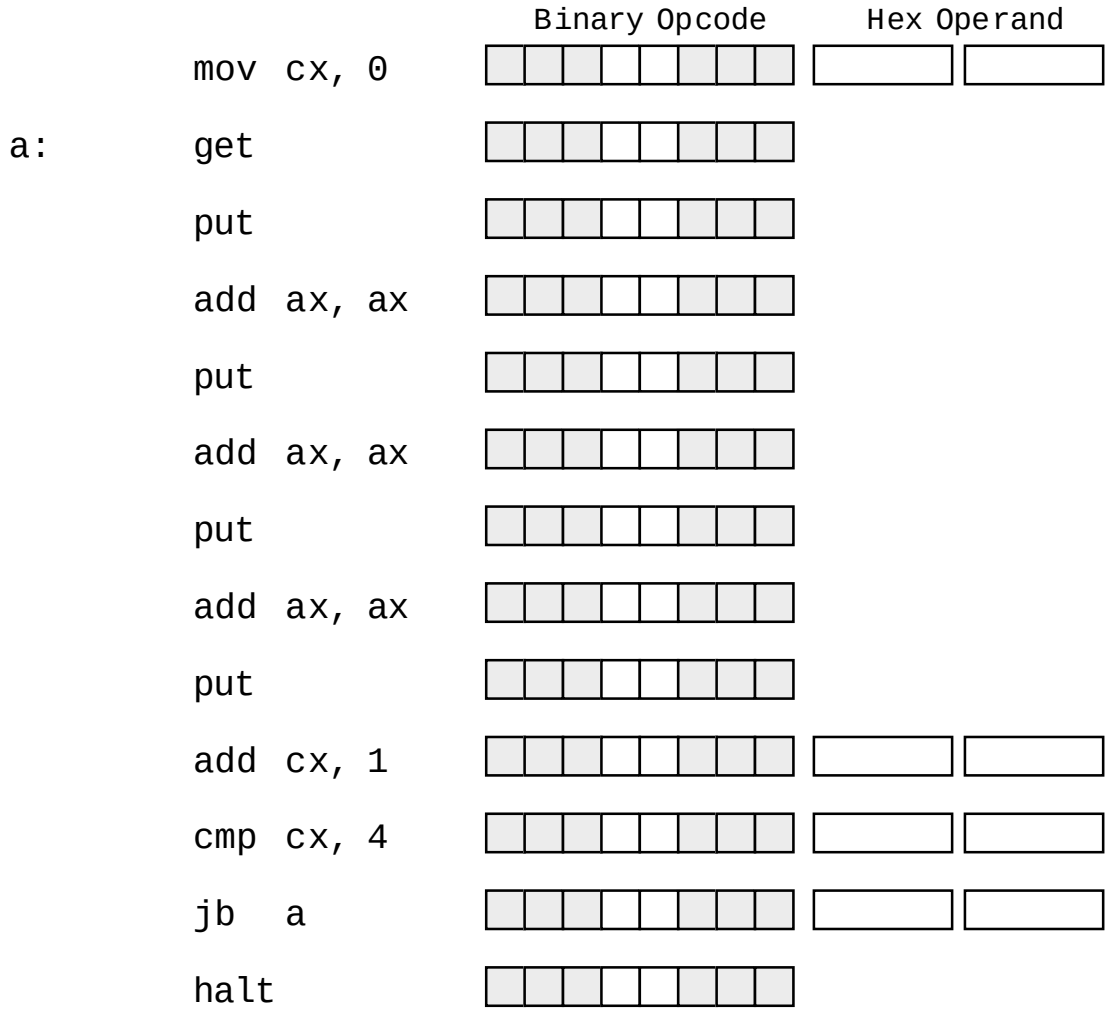
For your lab report: describe the output of the program whenever you press the interrupt button. Explain all the steps you would need to follow to place the interrupt service routine at address 2000h rather than 100h.

For additional credit: write your own interrupt service routine that does something simple. Run the main program and press the interrupt button to test your code. Verify that your ISR works properly.

3.6.6 Machine Language Programming & Instruction Encoding Exercises

To this point you have been creating machine language programs with SIMx86's built-in assembler. An assembler is a program that translates an ASCII source file containing textual representations of a program into the actual machine code. The assembler program saves you a considerable amount of work by translating human readable instructions into machine code. Although tedious, you can perform this translation yourself. In this exercise you will create some very short *machine language* programs by encoding the instructions and entering their hexadecimal opcodes into memory on the memory screen.

Using the instruction encodings found in Figure 3.19, Figure 3.20, Figure 3.21, and Figure 3.22, write the hexadecimal values for the opcodes beside each of the following instructions:



You can assume that the program starts at address zero and, therefore, label "a" will be at address 0003 since the `mov cx, 0` instruction is three bytes long.

For your lab report: enter the hexadecimal opcodes and operands into memory starting at location zero using the Memory editor screen. Dump these values and include them in your lab report. Switch to the Emulator screen and disassemble the code starting at address zero. Verify that this code is the same as the assembly code above. Print a copy of the disassembled code and include it in your lab report. Run the program and verify that it works properly.

3.6.7 Self Modifying Code Exercises

In the previous laboratory exercise, you discovered that the system doesn't really differentiate data and instructions in memory. You were able to enter hexadecimal data and the x86 processor treats it as a sequence of executable instructions. It is also possible for a program to store data into memory and then execute it. A program is *self-modifying* if it creates or modifies some of the instructions it executes.

Consider the following x86 program (EX6.x86):

```
        sub     ax, ax
        mov     [100], ax

a:      mov     ax, [100]
        cmp     ax, 0
        je      b
        halt

b:      mov     ax, 00c6
        mov     [100], ax
        mov     ax, 0710
        mov     [102], ax
        mov     ax, a6a0
        mov     [104], ax
        mov     ax, 1000
        mov     [106], ax
        mov     ax, 8007
        mov     [108], ax
        mov     ax, 00e6
        mov     [10a], ax
        mov     ax, 0e10
        mov     [10c], ax
        mov     ax, 4
        mov     [10e], ax
        jmp     100
```

This program writes the following code to location 100 and then executes it:

```
        mov     ax, [1000]
        put
        add     ax, ax
        add     ax, [1000]
        put
        sub     ax, ax
        mov     [1000], ax
        jmp     0004                ;0004 is the address of the A: label.
```

For your lab report: execute the EX7.x86 program and verify that it generates the above code at location 100.

Although this program demonstrates the principle of self-modifying code, it hardly does anything useful. As a general rule, one would not use self-modifying code in the manner above, where one segment writes some sequence of instructions and then executes them. Instead, most programs that use self-modifying code only modify existing instructions and often only the operands of those instructions.

Self-modifying code is rarely found in modern assembly language programs. Programs that are self-modifying are hard to read and understand, difficult to debug, and often unstable. Programmers often resort to self-modifying code when the CPU's architecture lacks sufficient power to achieve a desired goal. The later Intel 80x86 processors do not lack for instructions or addressing modes, so it is very rare to find 80x86 programs that use self-modifying code[23]. The x86 processors, however, have a very weak instruction set, so there are actually a couple of instances where self-modifying code may prove useful.

A good example of an architectural deficiency where the x86 is lacking is with respect to subroutines. The x86 instruction set does not provide any (direct) way to call and return from a subroutine. However, you can easily simulate a call and return using the jmp instruction and self-modifying code. Consider the following x86 "subroutine" that sits at location 100h in memory:

```
; Integer to Binary converter.
; Expects an unsigned integer value in AX.
; Converts this to a string of zeros and ones storing this string of
; values into memory starting at location 1000h.

        mov     bx, 1000            ;Starting address of string.
        mov     cx, 10              ;16 (10h) digits in a word.
a:      mov     dx, 0               ;Assume current bit is zero.
        cmp     ax, 8000            ;See if AX's H.O. bit is zero or one.
        jb      b                   ;Branch if AX'x H.O. bit is zero.
        mov     dx, 1               ;AX's H.O. bit is one, set that here.
b:      mov     [bx], dx            ;Store zero or one to next string loc.
        add     bx, 1               ;Bump BX up to next string location.
        add     ax, ax              ;AX = AX *2 (shift left operation).
        sub     cx, 1               ;Count off 16 bits.
        cmp     cx, 0               ;Repeat 16 times.
        ja      a
        jmp     0                   ;Return to caller via self-mod code.
```

The only instruction that a program will modify in this subroutine is the very last jmp instruction. This jump instruction must transfer control to the first instruction beyond the jmp in the calling code that transfers control to this subroutine; that is, the caller must store the return address into the operand of the jmp instruction in the code above. As it turns out, the jmp instruction is at address 120h (assuming the code above starts at location 100h). Therefore, the caller must store the return address into location 121h (the operand of the jmp instruction). The following sample "main" program makes three calls to the "subroutine" above:

```
        mov     ax, 000c            ;Address of the BRK instr below.
        mov     [121], ax           ;Store into JMP as return address.
        mov     ax, 1234            ;Convert 1234h to binary.
        jmp     100                 ;"Call" the subroutine above.
        brk                         ;Pause to let the user examine 1000h.

        mov     ax, 0019            ;Address of the brk instr below.
        mov     [121], ax
        mov     ax, fdeb            ;Convert 0FDEBh to binary.
        jmp     100
        brk

        mov     ax, 26              ;Address of the halt instr below.
        mov     [121], ax
        mov     ax, 2345            ;Convert 2345h to binary.
        jmp     100

        halt
```

23. Many viruses and copy protection programs use self modifying code to make it difficult to detect or bypass them.

Load the subroutine (EX7s.x86) into SIMx86 and assemble it starting at location 100h. Next, load the main program (EX7m.x86) into memory and assemble it starting at location zero. Switch to the Emulator screen and verify that all the return addresses (0ch, 19h, and 26h) are correct. Also verify that the return address needs to be written to location 121h. Next, run the program. The program will execute a brk instruction after each of the first two calls. The brk instruction pauses the program. At this point you can switch to the memory screen at look at locations 1000-100F in memory. They should contain the pseudo-binary conversion of the value passed to the subroutine. Once you verify that the conversion is correct, switch back to the Emulator screen and press the Run button to continue program execution after the brk.

For your lab report: describe how self-modifying code works and explain in detail how this code uses self-modifying code to simulate call and return instructions. Explain the modifications you would need to make to move the main program to address 800h and the subroutine to location 900h.

For additional credit: Actually change the program and subroutine so that they work properly at the addresses above (800h and 900h).

3.7 Programming Projects

Note: You are to write these programs in x86 assembly language code using the SIMx86 program. Include a specification document, a test plan, a program listing, and sample output with your program submissions

1) The x86 instruction set does not include a multiply instruction. Write a short program that reads two values from the user and displays their product (hint: remember that multiplication is just repeated addition).

2) Create a callable subroutine that performs the multplication inproblem (1) above. Pass the two values to multiple to the subroutine in the ax and bx registers. Return the product in the cx register. Use the self-modifying code technique found in the section “Self Modifying Code Exercises” on page 136.

3) Write a program that reads two two-bit numbers from switches (FFF0/FFF2) and (FFF4/FFF6). Treating these bits as logical values, your code should compute the three-bit sum of these two values (two-bit result plus a carry). Use the logic equations for the full adder from the previous chapter. *Do not simply add these values using the x86 add instruction.* Display the three-bit result on LEDs FFF8, FFFA, and FFFC.

4) Write a subroutine that expects an address in BX, a count in CX, and a value in AX. It should write CX copies of AX to successive words in memory starting at address BX. Write a main program that calls this subroutine several times with different addresses. Use the self-modifying code subroutine call and return mechanism described in the laboratory exercises.

5) Write the generic logic function for the x86 processor (see Chapter Two). Hint: add ax, ax does a shift left on the value in ax. You can test to see if the high order bit is set by checking to see if ax is greater than 8000h.

6) Write a program that reads the generic function number for a four-input function from the user and then continually reads the switches and writes the result to an LED.

7) Write a program that scans an array of words starting at address 1000h and memory, of the length specified by the value in cx, and locates the maximum value in that array. Display the value after scanning the array.

8) Write a program that computes the two’s complement of an array of values starting at location 1000h. CX should contain the number of values in the array. Assume each array element is a two-byte integer.

9) Write a “light show” program that displays a “light show” on the SIMx86’s LEDs. It should accomplish this by writing a set of values to the LEDs, delaying for some time