# System Organization — Lab Manual, Chapter Three

A good assembly language programmer needs to know a little more than an assembly language instruction set. An in-depth understanding of the underlying computer architecture is also necessary. Unless you know the internal operation of the instructions and how these internal operations interact, you cannot write the fastest possible assembly language code. Likewise, without knowing how a machine encodes instructions, you will not be able to predict if one instruction sequence is shorter or longer than another.

In this laboratory you will learn how to use a rudimentary *debugger* to create, test, and execute short *machine language programs*. You will also explore *cycle counting* and *byte-counting* program optimization methods. These are skills which are important when working with complex assembly language programs.

## 3.1 Debuggers and SIM886

A debugger is a program which lets you display and modify different memory locations, execute instructions, display machine registers, and perform other common operations necessary to test and correct machine language programs. The correct use of a debugger will dramatically reduce the time you spend testing and correcting assembly language programs. In this laboratory you will use a very simple machine language debugger for the x86 processors. This will provide a gentle introduction to the CodeView debugger which Chapter Four describes.

Before describing the debugger itself, a discussion of the x86 simulator is necessary. Since the x86 processors are hypothetical, there isn't any hardware on which you can run an x86 program. The x86 simulator (SIM886) is a piece of software which simulates an x86 processor. This program fetches x86 opcodes from memory (the 80x86's) and then executes a sequence of instructions which emulates the behavior of those instructions.

The SIM886 program uses integer variables to hold the values of the 886 registers and memory locations. For all intents and purposes, you could not tell the difference between the SIM886 program and an actual 886 processor (if one existed). This simulation technique is not limited to hypothetical processors, by the way. SoftPC and other programs use this very technique to emulate an 80x86 processor on Macintosh, NeXT, and other computer systems.

The SIM886 program can control the LED and switch circuitry you built for the lab exercises in Chapter Two. When you run SIM886, it defaults to LPT1:. If you use the circuitry and you install it in a parallel port other than LPT1:, you should specify the port name as a command line argument to the SIM886 program:

```
c:> sim886  lpt1:
c:> sim886  lpt2:
c:> sim886  lpt3:
```

### 3.1.1 Displaying and Entering Memory Values

The SIM886 program simulates the 64K address space of the 886 processor using a 64K array in the 80x86 memory space. Whatever would normally appear at memory location zero on the 886 processor winds up at index zero in that array. Like any decent debugger, the SIM886 program lets you display and enter memory values. When you first run the SIM886 program, it initializes all bytes in the 886 memory space so that they contain 0E0h (which turns out to be the opcode of the 886 HALT instruction). You can verify this by using the SIM886 "D" (for Dump or Display). This command takes the form:

D *address*

This command displays up to 16 bytes of data starting at the specified address. For example, the command "D 1000" displays 16 bytes starting at address 1000h[1] as follows:

```
1000: E0 E0 E0 E0  E0 E0 E0 E0    E0 E0 E0 E0  E0 E0 E0 E0
```

1. All constant values in the debugger are hexadecimal values. You do not need to (in fact, you cannot) place the "H" suffix after the value.

The first E0 above is the value for location 1000h, the second E0 is the value contained in location 1001h, …, the last E0 displayed above is for location 100Fh. Note that there are extra spaces between the values for 1003h/1004h, 1007h/1008h, and 100Bh/100Ch. This makes locating a specific value on the line a little easier.

The **D**ump command is very useful for displaying the values of variables, arrays, or even machine code (binary representations for instructions). However, you must always keep one important point in mind– the **D**ump command displays values across the line from a low address to a high address. This seems reasonable enough. But note that word values, which the 886 processors stores in two consecutive bytes with the low-order byte at the lowest address and the high-order byte at the highest address, will appear backwards on the dump display. If the word at address 1000h contains 1234h and you use the **D**ump command, you'll see something like the following:

```
1000: 34 12 xx xx  xx xx xx xx    xx xx xx xx  xx xx xx xx
```

You must remember to mentally swap the values around when reading them from the screen. A very common mistake beginners make is to forget to swap the bytes displayed by the debugger.

The **D**ump command lets you view the contents of memory, the **E**nter command lets you change the contents of one or more memory locations. The format for the **E**nter command is

```
E address
```

This instructs the SIM886 program to begin accepting input values from the keyboard. Immediately after you enter the command "E 1000" SIM886 responds with

```
1000(E0): _
```

The first four digits are the address you plan to modify (in this example, address 1000h). The value in the parentheses is the current value for that location. The value E0 above is what you would expect immediately after loading the SIM886 program into memory. Of course, if you've previously changed the contents of memory location 1000h, it will display the current value rather than E0.

To change the contents of the specified location, simply enter an eight-bit hex constant (without the trailing "h", remember SIM886 assumes all values are hexadecimal) and press [enter]. If the value you enter is outside the range 0..FF then SIM886 will print an appropriate error message. If the value begins with a non-hexadecimal character, SIM886 will simply set the location to zero. The x86 processors only support 16-bit registers; therefore, most values you input will be 16 bits. *This means you need to enter two bytes into memory for each value.* SIM886 provides eight bit entry mainly for entering instruction opcodes.

After you hit [enter] SIM886 will display "1001(E0): _" and prompt you to enter a new value for location 1001h. Each time you enter a value, SIM886 will prompt you to enter a value for the next location. When you are through entering data (or if you want to enter data at some non-consecutive address), simply hit [enter] without typing anything else.

**Note: Don't forget to enter word values with the L.O. byte first and the H.O. byte second.**

**3.1 How would you display the value of memory location 18FAh in the 886 memory space using the SIM886 debugger?**

______________________________________________________________

**3.2 What sequence of commands would you use to enter the word value 9837h to memory starting at location 8000h?**

______________________________________________________________

One last point concerning the **E**nter command which is worth mentioning: if you type a valid hexadecimal value followed by a space, SIM886 ignores the data on the line following the hexadecimal value. This allows you to type in values like the following:

```
E 1000
1000(00): E0 HALT instruction[enter]
1001(00): [enter]
```

The command above enters E0 into memory location 1000h. SIM886 ignores everything on the line following the E0 (in this case, the "HALT instruction" comment).

### 3.1.2 Disassembling (Unassembling) 886 Instructions

Had you not been told that "E0" was the binary opcode for the HALT instruction you might not have ever guessed the significance of this value. All 886 instructions appear in memory as numeric values. Trying to decipher the meaning of these numeric values is quite tedious and error-prone. Fortunately, this isn't something you'll really need to do because SIM886 can do this for you automatically. The **U**nassemble command will disassemble bytes in memory producing human-readable forms of the instructions.

The **U**nassemble command uses the syntax:

```
U address
```

where *address* is the hexadecimal address of the first byte of the instruction sequence you wish to have disassembled. Since the 886 stores instruction codes in successive bytes in memory, the **U**nassemble command will disassemble sequential instructions. This command disassembles 15 instructions starting at the address you specify. Since instructions on the 886 are one or three bytes long, the number of bytes disassembled may be as few as 15 bytes or as many as 45 bytes.

3.3 **If you wanted to enter the three byte instruction "ADD CX, 1" (the opcode bytes are 57 01 00) at address 1000h, what SIM886 command(s) would you use?**

________________________________________________

3.4 **What SIM886 command would you use to view the instruction in a human readable form after entering it into the machine?**

________________________________________________

### 3.1.3 Displaying and Modifying CPU Registers

The **R**egister command ("R") displays the current values of the 886 registers. It will display the values for AX, BX, CX, DX, and IP. It also disassembles the current instruction pointed at by IP. Unlike most debuggers, SIM886 does *not* let you modify the values of the 886's registers. In a real assembly language debugger this would be a problem. However, the programs you will be writing in 886 machine code will be very short and simple. You will not miss the ability to modify the registers. This command is mainly for tracing through programs and watching the results produced.

As it turns out, you won't use the **R**egister command to display the registers very often. The **T**race command displays the 886's registers after it steps through each instruction. You'll typically use the **R**egister command only after a disassembly or memory dump and the previous register values have scrolled off the screen.

The 886 register set usually contains zero upon initial entry into the SIM886 program. After you execute some 886 instructions the register values may change. The register set always maintains the values last stored into the registers. If you want to manually change a register value, type one of the following commands

```
RAX value
RBX value
RCX value
RDX value
```

These commands let you assign a hexadecimal value to the 886 registers.

## 3.1.4 Executing 886 Instructions

The SIM886 **T**race instruction *single steps* through a single instruction. The syntax for this instruction is

```
T address
```

SIM886 executes a single instruction at the specified address. After executing the instruction, SIM886 displays the registers and disassembles the next instruction in memory following the executed instruction.

Another command you can use to execute 886 instructions is the Go command. Go begins execution which stops when you press control-C or the SIM886 program encounters a HALT instruction:

```
G address
```

begins execution at the specified address

## 3.1.5 File I/O Commands

SIM886 provides three commands to let you read data from a text file and write data to a text file on the disk. The **I**nput command takes the following form:

```
I filename
```

*Filename* must be a valid DOS pathname. The **I**nput command tells SIM886 to read the next set of keyboard commands directly from the file rather than the keyboard. When SIM886 encounters the end of this file, it reverts input back to the keyboard. The primary purpose for the **I**nput command is to allow you to create an 886 assembly language program with a text editor (such as DOS' EDIT program) and then read this file into SIM886. A typical source file you would create with an editor might look like the following:

```
A 0
load ax, 0
load bx, 1
add ax, bx

u 0
```

Note the "A 0" command as the first line of this file. Remember, after processing the **I**nput command, SIM886 is going to try to read another command from the standard input. Since the input is coming from the file, the file must contain the command that activates the SIM886 assembler. The "u 0" command instructions SIM886 to unassemble this code after assembling it.

The other two file commands in SIM886 let you capture all output that goes to the screen and send a copy of it to a DOS text file. This provides a convenient way to capture the output of a SIM886 session without using the Ctrl P option to send the output to a printer. This allows you, for example, to create a text file transcription you can import into the word processor you're using to create your lab report.

To begin capturing data, use the "C *filename*" command. From that point forward, all data written to the screen is also written to the output file. To turn off the capture operation, use the "X" command.

## 3.1.6 Miscellaneous SIM886 Commands and Other Notes

If you forget one of the SIM886 commands, you can get an instant refresher using the SIM886 *help* command. Pressing the question mark ("?") displays a list of the available commands with a brief description of their syntax and operation. Furthermore, if any new commands have been added since the publication of this manual, or if your local installation has added some special commands, the help command will tell you about them.

When you are done using SIM886, you can return to MS-DOS using the **Q**uit command. Simply press "Q" followed by enter at the SIM886 prompt to quit the program. Note that if you restart the SIM886 program after quitting it, you lose everything in the 886 memory space. You will have to reenter such values if you need them.

On all of the SIM886 commands, the address following the command is optional. If you do not supply an address, SIM886 will look at the last time you used that command and supply the next available address for the current command. For example, if you use the command "D 1000" to dump memory locations 1000…100F and then press "D enter", SIM886 will automatically supply the value 1010 for you.

3.1 D 18FA

SIM886 remembers the last address for each specific command. If you enter the commands "D 1000" followed by "U 2000" and then press "D enter", SIM886 will display locations 1010…101F, *not* the bytes following the last disassembled instruction. Likewise, were you to enter the command "U œ" at this point, it would continue the disassembly after the previous instruction, it would not begin the disassembly at address 1020. SIM886 maintains separate *current address* values for the D, E, U, and X commands.

3.2 E 8000
37
98

Another useful shortcut is the enter command. If you press the enter key by itself on a SIM886 command line, it repeats the last instruction at the new address. For example, if you enter the command "D 1000 enter" and then press enter, SIM886 will repeat the memory dump command beginning at location 1010. Repeatedly pressing enter will dump memory locations 1020…102F, 1030…103F, etc. This "repeat last command" operation is particularly useful with the **T**race command. After you use the "T" command to execute the first instruction, repeatedly pressing enter will single step through the following instructions.

3.3 E 1000
57
01
00

The "W *count*" command lets you specify the number of *wait states* SIM886 will use on each memory access. By default, SIM886 uses zero wait states so accessing memory at an even address requires only one clock cycle. You can change the number of wait states using this command. Like all SIM886 commands, the numeric value you specify is a *hexadecimal* number. Don't forget this if you want to specify more than nine wait states. If you enter the "W" command without any parameters, SIM886 defaults back to zero wait states. If you want to see the current number of wait states that SIM886 is using, use the **R**egister command to display the register values. The "WS" register specifies the number of wait states on each memory access.

3.4 U 1000

SIM886 maintains a global clock cycle counter. After executing each instruction SIM886 adds the number of clock cycles for that instruction to this global counter. This lets you trace through a sequence of instructions and determine how many clock cycles occurred during the execution of those instructions. You can view the global clock cycle counter by using the **R**egister command. The "Z" (for **Z**ero) command resets this global clock cycle counter to zero. If you want to time a sequence of instructions, you should reset the global cycle counter before executing those instructions.

## 3.2 Machine Language vs. Assembly Language

As you may have noticed, there isn't a command in SIM886 to enter 886 instructions into memory. You can execute instruction using the **T**race command, but there isn't a command (described yet) to let you type something like "LOAD AX,0" into the debugger. Well, it turns out that the 886 processor (hypothetical or otherwise) does *not* execute instructions like "LOAD AX,0". Instead, it executes numeric encodings of these instructions. The "LOAD AX,0" instruction actually consists of three one byte values: 07 00 00. These three values tell an x86 processor to load the AX register with zero. These numeric forms of the instruction are *machine code*– instructions specifically for the machine. The human-readable form, "LOAD AX,0", is *assembly language*[2]. As you can probably tell, it's much easier to work in assembly language than in machine language. Although "LOAD AX,0" might not be the most understandable command you've ever seen, it's probably a lot easier to read and understand than " 07 00 00".

2. The term "assembly language" came about because the process of converting assembly code to machine code traditionally consists of building (or *assembling*) binary machine code instructions from various bit patterns which make up the instruction, operands, and addressing modes.

Most of the time you will be working in assembly language. This is, after all, an assembly language text, not a machine language text. However, it's definitely worthwhile to spend a little time working with machine language. This achieves three things: (1) it gives you an appreciation for what the assembler does for you; (2) many debugging problems require that you work at the machine level; and, most importantly, (3) working at the machine level forces you to take a close look at the underlying architecture, which you might be able to ignore at the assembly level.

The 80x86 processor family is quite complex and difficult for beginners to understand at the machine level. Indeed, that's the whole reason for introducing the x86 hypothetical processors in this text. The x86 family is very simple. Its design helps demonstrate important architectural features without being laden down with 1976 design decisions[3]. Nonetheless, Much of what you learn concerning the 886 applies almost directly to the 80x86 family as well.

**3.5 Given that the machine code for "LOAD AX,0" is 07 00 00, what SIM886 command would you use to enter this instruction into memory at address 2000h?**

______________________________________________________________

## 3.3 A Review of the 886 Instruction Set

The x86 processor family supports eleven different instructions. They are *LOAD, STORE, ADD, SUB, IFEQ, IFLT, IFGT, HALT, GOTO, GET,* and *PUT.* With the exception of the HALT instruction, all of these instructions require one or more operands.

The LOAD, STORE, ADD, and SUB instructions all have identical operands. These instructions take the form:

```
        instr       reg,  operand
```

where "instr" is any of "LOAD", "STORE", "ADD", or "SUB"; "reg" is any of "AX", "BX", "CX", or "DX"; and operand is one of "AX", "BX", "CX", "DX", "[BX]", "const[BX]", "[const]", or "const".

The LOAD instruction makes a copy of the second operand and places it in the first operand. This instruction overwrites the value of the first operand and does not affect the value of the second operand. You generally use this instruction to copy a value from memory into one of the four 886 general purpose registers, although you can use it to copy data from one register to another as well.

The second operand (the source of the data to copy to the first operand) can be one of the four registers, a memory address, or a constant. Quickly, here's what each of the forms of this instruction do:

```
LOAD ax, ax       ;Produces no observable results!
LOAD ax, bx       ;Copies value in BX into the AX reg.
LOAD ax, cx       ;Copies value in CX into the AX reg.
LOAD ax, dx       ;Copies value in DX into the AX reg.
```

(Note that you may substitute BX, CX, or DX for the first operand above to load the specified value into BX, CX, or DX rather than AX. For example "LOAD bx, ax" copies the value in the AX register into BX.)

```
LOAD  ax, [bx]    ;Copies data from memory location "bx"
                  ; into ax.
LOAD  ax, nn[bx]  ;Copies data from memory location "bx +
                  ; nn" into ax. "nn" represents a 16-bit
                  ; constant.
LOAD  ax, [nn]    ;Copies data from memory location "nn"
                  ; into ax. "nn" represents a 16-bit
                  ; constant.
LOAD  ax, nn      ;Copies 16-bit constant "nn" into ax.
```

These last four forms of the LOAD instruction probably deserve a little more explanation. The first three of these introduce different *memory addressing modes.* That is, they provide different ways of accessing memory. The first of these, "[BX]" is typically referred to as the *indirect addressing mode.* This addressing mode does not copy the value of BX into AX. That would be *direct* addressing. Instead, BX contains a value which the instruction uses as the memory address

3. If you hadn't guessed, the 8086 was first developed in 1976. Many of the complexities of the 80x86 family stem from the fact that the original 8086 designers didn't expect their processor to last so long in the marketplace.

of the actual value to fetch. For example, if BX contained 1800h, “LOAD ax, [bx]” would load ax with the word value beginning at memory address 1800h.

The second memory addressing mode above is the *indexed addressing mode*. As you’ll see in Chapter Four, this addressing mode is particularly useful for accessing elements of arrays and records. This address mode adds the value contained in BX with the 16-bit constant specified by “nn”. The 886 uses their sum as the *effective address* and loads the specified register (AX in this case) with the 16-bit value starting at that address.

The third memory addressing mode, “[nn]”, is the *direct addressing mode*. The second operand is just a 16-bit constant which provides the memory address to fetch and load into the destination register (AX in this case). You would use this addressing mode, for example, to access 16-bit variables in memory.

The last addressing mode isn’t really a memory addressing mode, it’s the immediate addressing mode which lets you load constants into a register. The “LOAD ax, nn” instruction (assuming “nn” represents a 16-bit hexadecimal constant) load the AX register with the value “nn”. For example, “LOAD ax, 0” loads the AX register with zero.

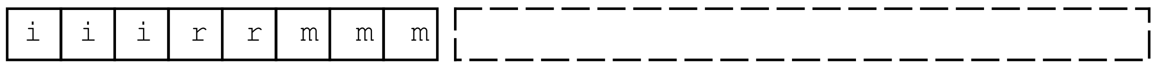
As mentioned earlier, the STORE, ADD, and SUB instructions allow the same operands as the LOAD instruction. The STORE instruction is the converse operation to LOAD. It stores the first operand to the location specified by the second operand. Note that the STORE instruction does *not* allow the immediate addressing mode (it doesn’t make sense to store a register into a constant). ADD and SUB, however, allow all addressing modes. Note that certain LOAD and STORE instructions perform the same operation. “LOAD ax, bx”, for example, does the same thing as “STORE bx, ax”.

The ADD instruction adds the value specified by the second operand to the first operand leaving the result in the first operand. The SUB instruction subtracts the second operand from the first operand leaving the result in the first operand. Both instructions allow all eight possible addressing modes for the second operand (four register modes, three memory modes, and the immediate mode).

The *IFEQ, IFGT,* and *IFLT* instructions all take three operands:

```
IFEQ  reg1, reg2, nn
IFGT  reg1, reg2, nn
IFLT  reg1, reg2, nn
```

$reg_1$ and $reg_2$ represent one of the four 886 registers: AX, BX, CX, or DX. “nn”, as usual, represents a 16-bit constant. These instructions compare $reg_1$ to $reg_2$. For IFEQ, if the two registers are



| i i i | r r | mmm |
|---|---|---|
| 000 = load | 00 = AX | 0 0 0 = AX |
| 001 = store | 01 = BX | 0 0 1 = BX |
| 010 = add | 10 = CX | 0 1 0 = CX |
| 011 = sub | 11 = DX | 0 1 1 = DX |
| 100 = ifeq | | 1 0 0 = [BX] |
| 101 = iflt | | 1 0 1 = xxxx[BX] |
| 110 = ifgt | | 1 1 0 = [xxxx] |
| 111 = special | | 1 1 1 = constant |

Note, this 16 bit operand is present only if mmm is 101, 110, or 111 or if iii is 100, 101, 110, or 111; if iii is 100, 101, or 110, then mmm may only be 000, 001, 010, or 011. If iii is 111 then the instruction uses a special encoding described elsewhere.

X86 Instruction Encoding

equal, the 886 loads the IP register with the constant. If the two registers are not equal, the 886 ignores the constant. IFGT and IFLT work in a similar manner except they load IP with the constant if $reg_1$ is greater than $reg_2$ or $reg_1$ is less than $reg_2$, respectively. Note that loading the IP (instruction pointer) register with "nn" will interrupt the normal flow of execution. Instead of executing the instruction following IFEQ, IFGT, or IFLT, the 886 will execute the instruction at address "nn" if the IFxx instruction loads IP with "nn". For this reason, these are *transfer of control* instructions.

The GET and PUT instructions let you perform simple I/O when running 886 programs. They both accept a single operand which is identical to the second operand of the STORE instruction (that is, a register or a memory addressing mode).

The GET instruction stops execution and prompts you to enter a value (in hexadecimal) from the keyboard. The PUT instruction outputs the specified value to the screen. The GET instruction is particularly useful for loading different values into registers while the program is running. Although the GET instruction allows a memory addressing mode, using the GET instruction to read a value into a memory location is of marginal utility. After all, you can use the SIM886 **E**nter instruction to accomplish the same thing. Of course, were you to modify SIM886 to execute more than one instruction at a time, this instruction would be more useful.

The PUT instruction outputs the specified register or memory operand. Since SIM886 displays the contents of all 886 registers after executing each instruction, using PUT to output a register value won't buy you much. It is quite handy for output memory locations, however (this saves you the effort of using the SIM886 **D**ump command).

The GOTO instruction must be followed by a 16-bit constant. This instruction loads the IP register with the value of the constant. Like the IFxx instructions, the GOTO instruction is a transfer of control instruction. The IFxx instructions are *conditional transfer instructions* since they transfer control (or not) based on some condition being true. The GOTO instruction is an *unconditional transfer instruction* since it transfers control regardless of any condition which may or may not exist.

The HALT instruction is the only 886 instruction which doesn't require any operands. When SIM886 encounters this instruction it stops processing. The HALT instruction does not update the IP to point at the next instruction. If you execute HALT and immediately specify the e**X**ecute instruction again (without specifying a new address) will execute the same HALT instruction.

**3.6 How could you copy the contents of memory locations 1000/1001h to memory locations 8000/8001h using 886 instructions? (hint: feel free to use more than one instruction if necessary)**

________________________________________________________________

________________________________________________________________

________________________________________________________________

**3.7 Write a short 886 program which inputs a value from the keyboard and sets AX to zero if the value is less than 8000h or sets AX to 1 if the input value is greater than or equal to 8000h, then halts.**

________________________________________________________________

________________________________________________________________

________________________________________________________________

________________________________________________________________

________________________________________________________________

________________________________________________________________

________________________________________________________________

________________________________________________________________

________________________________________________________________

________________________________________________________________

## 3.4 Converting 886 Programs to Machine Code

3.5 E 2000
07
00
00

The preceding section described the operation of the 886 processor in terms of assembly language. Alas, you've already been told that SIM886 doesn't support assembly language, only machine code. This means you've got to convert nice, readable (!), instructions like "LOAD AX,0" into sequences of numbers like "07 00 00" before you can enter the instruction into SIM886. The question is "How on Earth does someone memorize all the possible instruction codes?" The answer is quite simple– you don't. Instead, you *assemble* the instruction opcodes yourself using some relatively simple look ups.

The 886 processor uses a relatively simple instruction encoding scheme. All 886 instructions use the encodings below. All opcodes are one byte long. All instructions are one or three bytes long (the extra two bytes in the three-byte instructions hold the 16-bit constants appearing in 886 instructions). The opcode byte always appears first in memory (that is, at the lowest address). If a 16-bit constant follows the opcode, the L.O. byte of the constant comes first followed by the H.O. byte.

Bits zero through two of the opcode byte contain the addressing mode for the second operand. Bits three and four encode the value for the first register operand. Bits five through seven encode the instruction itself.

As simple as the 886 instruction encoding scheme is, there are still some inconsistencies of which you must be aware. For example, the 886 has eleven different instructions, but only three bits to encode the instructions. Three bits lets you specify eight different instructions, not eleven. To get around this, the 886 uses a special instruction encoding, iii=111 (the *special* entry above) to *extend* the instruction set. When the opcode field contains 111, the register field determinates the actual instruction. The for possible values are

```
00 - Halt          01 - Goto          10 - Get          11
- Put
```

Since these instructions use the register operand field to encode the operation, there is only one field left for the operand. Goto, get, and put only allow a single operand, so this works out well. Halt does not allow any operands, so if the H.O. five bits of the opcode contain 11100, SIM886 ignores the L.O. three bits (the second operand field).

Finally, consider the IFxx instructions. They have *three* operands. How does the encoding above handle them? Easy. The register and operand fields (bits 0..4) encode two registers. The third operand is a displacement, so these instructions always include a 16 bit displacement value after the opcode.

### 3.4.1 Encoding the LOAD, STORE, ADD, and SUB Instructions

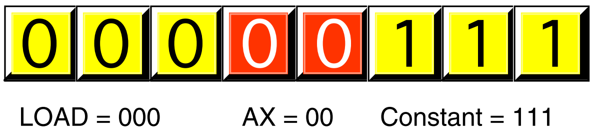
The LOAD, STORE, ADD, and SUB instructions are the easiest to handle, so they're the best place to start. With one single exception, these four instructions implement the encoding above exactly. The only exception is that the encoding above allows a machine code instruction of the form "27 00 00" which corresponds to a "STORE reg, immediate" instruction. Since there is no such thing as a store immediate instruction, we have a problem. To solve this problem the 886 adopts the same strategy as most processors– it treats this as an illegal instruction which programmers are not supposed to put into the program. Doing so may produce an error or unpredictable results.

To convert the "LOAD AX, 0" instruction to its equivalent machine code, the first step is to construct the opcode for this instruction:

- Since this is the LOAD instruction, bits five through seven of the opcode must be 000.

- The register operand (the first operand) is AX. The bit code for this operand is 00 (for bits three and four).
- Finally, this instruction uses the immediate (constant) addressing mode. Bits zero through two must contain 111 for the immediate addressing mode.
- Two additional bytes containing the immediate constant (0000h) must follow the opcode.

This yields the following opcode:



LOAD AX, 0   Encoding

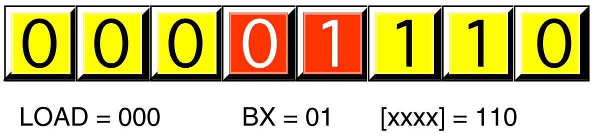
Two bytes of zeros, representing the 16-bit immediate constant, must follow the opcode low order byte first. Therefore the complete instruction is three bytes long and is 07 00 00. So now you can see where these "magical" numbers came from.

For LOAD, STORE, ADD, and SUB, if the addressing mode is 101, 110, or 111 (xxxx[bx]. [xxxx],. or constant) then there will be a 16-bit constant following the opcode (L.O. byte first). All other forms of these instructions are one byte long. Some additional examples will help cement this down.

Consider the "LOAD BX, [101F]" instruction. You build the opcode as follows:

- Since this is the LOAD instruction, bits five through seven of the opcode must be 000.
- The register operand (the first operand) is BX. The bit code for this operand is 01 (for bits three and four).
- Finally, this instruction uses the direct ([xxxx]) addressing mode. Bits zero through two must contain 110 for the immediate addressing mode.
- Two additional bytes, containing 101Fh, must follow the opcode (1F first, 10 second).

This yields the following opcode:



LOAD BX, [101F]   Encoding

**3.8 How many bytes long is this instruction?**________________________

**3.9 What is the hexadecimal machine code encoding for this instruction?**

__________________________________________________________________

**3.10 What will this instruction do when the 886 executes it?**

__________________________________________________________________

__________________________________________________________________

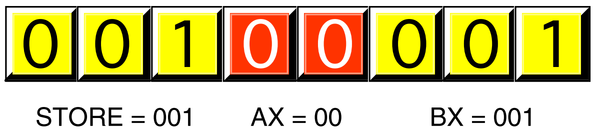
**3.11 What would the encoding be for "LOAD DX, [101F]"?**

__________________________________________________________________

Consider the "STORE AX, BX" instruction:

- Since this is the STORE instruction, bits five through seven of the opcode must be 001.
- The register operand (the first operand) is AX. The bit code for this operand is 00 (for bits three and four).
- This instruction uses the register/BX addressing mode. Bits zero through two must contain 001 for BX.

The opcode is



STORE AX, BX Encoding

**3.12 How many bytes long is this instruction?** ______________________

**3.13 What is the hexadecimal machine code encoding for this instruction?**

____________________________________________________________

**3.14 What will this instruction do when the 886 executes it?**

____________________________________________________________

____________________________________________________________

____________________________________________________________

**3.15 What is another instruction which performs the exact same operation as this instruction?**

____________________________________________________________

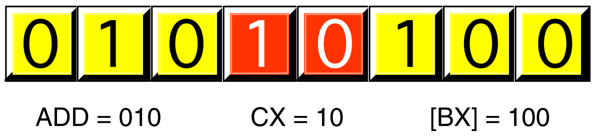
**3.16 What is the hexadecimal encoding for that other instruction?**

____________________________________________________________

Moving on to the ADD instruction, consider the encoding for the "ADD CX, [BX]" instruction:

- Bits five through seven of the opcode must be 010 for the ADD instruction.
- The register operand (the first operand) is CX. The bit code for this operand is 10 (for bits three and four).
- This instruction uses the [BX] addressing mode. Bits zero through two must contain 100 for [BX].

The opcode is



ADD CX, [BX] Encoding

**3.17 How many bytes long is this instruction?**

____________________________________________________________

**3.18 What is the hexadecimal machine code encoding for this instruction?**

____________________________________________________________

**3.19 What will this instruction do when the 886 executes it?**

____________________________________________________________

____________________________________________________________

____________________________________________________________

3.6 One way to do it:
```
Load ax, [1000]
Store ax, [8000]
```

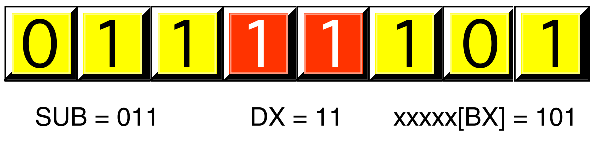
3.7 One way to do it:
```
       get bx
       load cx, 8000
       iflt bx, cx, Set0
       load ax, 1
       goto Done

Set0:  load ax, 0
Done:  halt
```

Note: Set0 and Done represent the addresses of the corresponding "labels" in this short program. If this code where assembled at address zero, Set0 would be equal to address 000Dh and Done would be equal to address 0010h.

Finally, consider the "SUB DX, 2002 [BX]" instruction. Its encoding is

- Bits five through seven of the opcode must be 011 for the sub instruction.
- The register operand (the first operand) is dx. The bit code for this operand is 11 (for bits three and four).
- This instruction uses the indexed (xxxx[BX]) addressing mode. Bits zero through two must contain 101 for xxxx[BX].
- Two displacement bytes containing the value 2002h (02 first, 20 second) must follow the opcode.

The opcode is



SUB DX, 2002 [BX] Encoding

**3.20 How many bytes long is this instruction? ________**

**3.21 What is the hexadecimal machine code encoding for this instruction?**

________________________________________________________________

**3.22 What will this instruction do when the 886 executes it?**

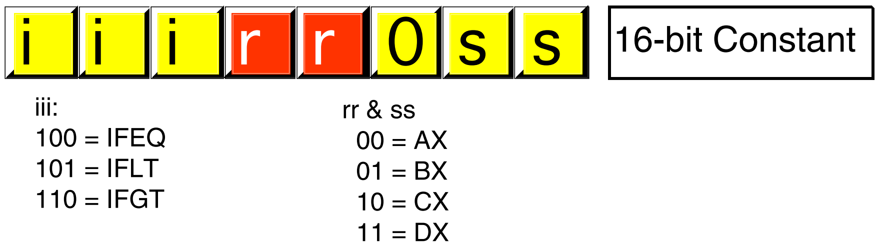
________________________________________________________________

________________________________________________________________

________________________________________________________________

## 3.4.2 Encoding the IFEQ, IFLT, and IFGT Instructions

The if*xx*[4] instruction encodings do not follow the same exact form as load, store, add, and sub. Considering that the if*xx* instructions have three operands, this shouldn't be too surprising. The generic encoding for the if*xx* instructions is



IFxx reg, reg, constant Encoding

Note that bit two *must be zero*. IFxx instructions containing a one in this bit position are illegal instructions. Also note that a 16-bit constant *always* follows the instruction, regardless of the opcode, hence all if*xx* instructions are three bytes long.

Bits three and four in the opcode supply the value for the first register operand. Bits zero and one supply the value for the second register operand. Since these are the only encodings available, the first two operands *must* be registers.

4. IFxx represents any of the three instructions IFEQ, IFLT, or IFGT.

The third operand is always a 16-bit constant, therefore the opcode doesn't need any extra bits to inform the x86 processor that the word following the opcode is a constant.

Remember, the x86 processors compare the two register operands and then transfer control to the address specified by the 16-bit constant if the contents of the registers are equal (`ifeq`), less than (`iflt`), or greater than (`ifgt`). Algorithmically, we can describe the operation of these instructions as follows:

```
ifeq reg$_1$, reg$_2$, constant

        if (reg$_1$ = reg$_2$) IP := constant
        else IP := IP + 3;
iflt reg$_1$, reg$_2$, constant

        if (reg$_1$ < reg$_2$) IP := constant
        else IP := IP + 3;
ifgt reg1, reg2, constant

        if (reg1 > reg2) IP := constant
        else IP := IP + 3;
```

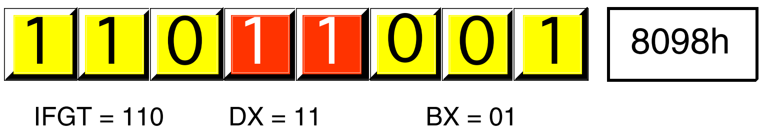
Were you to place a one in bit number two of the opcode, the x86 processor might confuse the 16-bit constant following the opcode with one of the 16-bit constants required by the memory or immediate addressing modes. This is why bit number two must always contain zero.

As a short digression, it's worth mentioning *why* the x86 processors require that the second operand be zero. Since we have the bit available, you might ask "Why not allow those extra addressing modes?" Of course, three of the extra addressing modes require their own 16-bit constant but that could be easily handled by extending the instruction to five bytes in such cases. However, doing so would complicate the instruction set, something which tends to defeat the purpose of the 886 in the first place. On the other hand, limiting the operands in this manner forces you to memorize which instructions allow memory/immediate operands and which do not. Processors which allow you to use the same operands for all instructions have *regular* or *orthogonal* instruction sets. The x86 processors do not allow all possible forms for each operand, so they are not completely orthogonal[5].

Now consider the encoding for the `ifgt dx, bx, 8098` instruction:

- Bits five through seven of the opcode must be 110 for the `ifgt` instruction.
- The first register operand is `dx`. The bit code for this operand is 11 (for bits three and four).
- The second register operand is `bx`. Bits zero through two must contain 010 for `bx`.
- Two displacement bytes containing the value 8098h(98 first, 90 second) must follow the opcode.

The opcode is



IFGT DX, BX, 8098h Encoding

5. It should be pointed at that there aren't *any* processors which have completely orthogonal instruction sets. Orthogonality is a measurement best described by terms like *almost orthogonal, mostly orthogonal, somewhat orthogonal,* and *non-orthogonal.*

3.8 3

3.9 0E 1F 10

3.10 Load the BX register with a copy of the data at memory locations 101F/1020.

3.11 1E 1F 10

3.12 1

3.13 21

3.14 3Copy the value in AX to the BX register.

3.15 Load BX, AX

3.16 28

3.17 1

3.18 54

3.19 Fetch the word from memory locations BX and BX+1 and place this value in CX.

The hexadecimal encoding for this instruction is D9 98 80.

**3.23 What is the encoding for the** `ifeq ax, bx, 1000h` **instruction?**

____________________________________________________________

**3.24 What is the encoding for the** `ifeq cx, ax, 1234h` **instruction?**

____________________________________________________________

**3.25 What is the encoding for the** `iflt bx, cx, 102h` **instruction?**

____________________________________________________________

**3.26 What is the encoding for the** `ifgt cx, ds, 8908h` **instruction?**

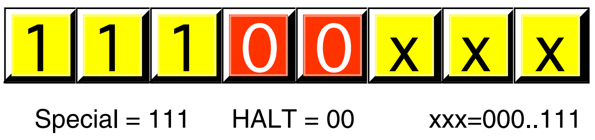
____________________________________________________________

**3.27 What is the encoding for the** `iflt dx, cx, 8880h` **instruction?**

____________________________________________________________

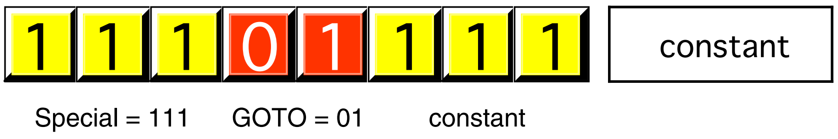
## 3.4.3 Encoding the HALT, GET, PUT, and GOTO Instructions

The HALT, GET, PUT, and GOTO instructions (the "special" instructions) use the following opcodes:



HALT Encoding

Note that the HALT instruction does not have any operands. Therefore, the x86 processors ignore the L.O. three bits of the opcode. Technically, 0E0h is the correct opcode for the HALT instruction and 0E1h…0E7h are illegal instructions. Note that the special instructions (HALT, GET, PUT, and GOTO) use bits three and four to encode the instruction.



GOTO Encoding

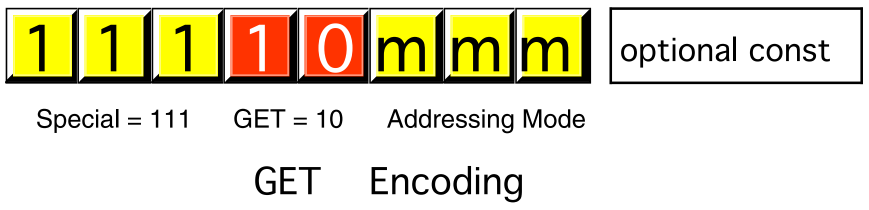
Like the HALT instruction, the L.O. three bits of the GOTO instruction must contain a specific value, "111" in this case[6]. In assembly form, the GOTO instruction always takes a single operand– a constant supplying the target address, e.g.,

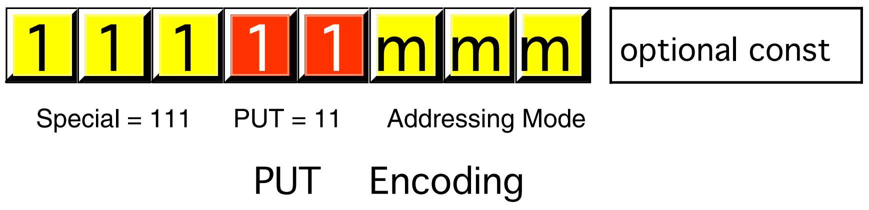
GOTO 1011h

6. Actually, the SIM886 program will allow all the various register, memory, and constant addressing modes. However, you will not need these modes and it isn't worth taking the time to explain how they operate. On the other hand, if you figure it out and use these modes in your lab report, that might be worth some extra credit!

The instruction code for this goto instruction is EF 11 10 (remember, L.O. byte of the constant comes first).

3.20 3





3.21 7D 02 20

The GET and PUT instructions both require a single operand which may be any valid x86 register or memory addressing mode. These instructions do not allow the immediate addressing mode[7]. Therefore, the "mmm" bits should never contain "111". Examples:

```
get ax       ;mmm = 000, so opcode is F0
put bx       ;mmm = 001, so opcode is F8
get [1000]   ;mmm = 110, so opcode is F6, const is 00 10
put [bx]     ;mmm = 100, so opcode is FC
get 1234[bx];mmm = 101, so opcode is F5, const is 34 12
put dx       ;mmm = 011, so opcode is FB
get cx       ;mmm = 010, so opcode is F2
put cx       ;mmm = 010, so opcode is FA
put 2[bx]    ;mmm = 101, so opcode is FD, const is 02 00
```

3.22 It will fetch the word from memory locations 2002+BX and 2003+BX and subtract this value from DX, leaving the difference in DX.

**3.28 What is the hexadecimal encoding for the "HALT" instruction? ________**

**3.29 What is the hexadecimal encoding for the "GOTO 1245h" instruction?**

________________________________________

**3.30 What is the hexadecimal encoding for the "PUT [BX]" instruction?**

________________________________________

**3.31 What is the hexadecimal encoding for the "PUT 15h[BX]" instruction?**

________________________________________

**3.32 What is the hexadecimal encoding for the "GET AX" instruction?**

________________________________________

**3.33 What is the hexadecimal encoding for the "GET [2025h]" instruction?**

________________________________________

**3.34 What is the hexadecimal encoding for the "PUT DX" instruction?**

________________________________________

7. Actually, the PUT instruction will accept the immediate mode, but this is of little value.

## 3.4.4 Creating Machine Language *Programs*

Thus far, you've seen how to convert a single x86 instruction into machine code. However, x86 programs generally contain many different x86 instructions. For example, consider the following short x86 program which reads five numbers from the user and displays their sum:

```
lp:          load  dx, 5       ;Repeat loop 5 times
             load  cx, 0       ;Loop counter
             load  ax, 0       ;Accumulate result here
             get   bx          ;Read value from user
             add   ax, bx      ;Add value to accumulator
             add   cx, 1       ;Bump loop counter by 1
             iflt  cx, dx, lp  ;Branch if not five times
             put   ax          ;Print result
             halt              ;Okay, we're done
```

Note: in the code above "lp" is a *symbolic label* which denotes the target of the `iflt` instruction. This is just a place holder. The actual target address is something we'll fill in momentarily. For now, just note that the `iflt` instruction will transfer control to the `get` instruction if `cx` is less than `dx` . Trace through the code above once by hand to convince yourself that it really will sum up five numbers and print the result.

To convert an assembly language program, like the one above, into machine language the first thing you need to do is decide *where* in memory you are going to place the program. By convention, most x86 programs start at location zero, though there is nothing stopping you from putting the program somewhere else in memory. Most of the examples in this text will start at memory location zero.

To convert the program above into machine code, you convert each individual instruction to its hexadecimal encoding. for the above, you would have:

```
             load  dx, 5       ;1F 05 00
             load  cx, 0       ;17 00 00
             load  ax, 0       ;07 00 00
lp:          get   bx          ;F1
             add   ax, bx      ;41
             add   cx, 1       ;57 01 00
             iflt  cx, dx, lp  ;B3 ?? ??
             put   ax          ;F8
             halt              ;E0
```

Note that the constant field of the IFLT instruction doesn't contain a particular value. This is because we need to fill in the address of the GET instruction at the "LP" label for this constant. To do that we need to store the bytes of each of these instructions into memory (starting at memory address zero):

3.23 81 00 10

3.24 90 34 12

3.25 AA 02 01

3.26 D3 08 89

3.27 BA 80 88

3.28 E0

3.29 EF 45 12

3.30 FC

3.31 F

3.32 F0

3.33 F6

3.34 FB

| Address | Code | Instruction |
|---|---|---|
| 0 | 1F | LOAD DX, 5 |
| 1 | 05 | |
| 2 | 00 | |
| 3 | 17 | LOAD CX, 0 |
| 4 | 00 | |
| 5 | 00 | |
| 6 | 07 | LOAD AX, 0 |
| 7 | 00 | |
| 8 | 00 | |
| 9 | F1 | GET BX |
| A | 41 | ADD AX, BX |
| B | 57 | ADD CX, 1 |
| C | 01 | |
| D | 00 | |
| E | B3 | IFLT CX, DX, 0009 |
| F | 09 | |
| 10 | 00 | |
| 11 | F8 | PUT AX |
| 12 | E0 | HALT |
| 13 | | |

After organizing the data in this fashion, it's much easier to determine how to fill in the constant for the IFLT instruction. By simply noting that the GET instruction appears at location 0009 we can supply this address as the 16-bit constant operand for the IFLT instruction. In this table each line represents one byte of code. Instructions which consume three bytes take up three lines in the table. There are a couple blank tables like this one provided in this lab manual so you can easily create your own x86 machine language programs. Feel free to make copies of these tables and use them for all your machine language programs.

**3.35 Once you've converted an x86 assembly language program into machine code, how would you enter this machine code into the SIM886 program?**

______________________________________________

**3.36 Suppose you were to enter the exact machine code in this example starting at memory location 1000h. Would the program still work? Why or why not?**

________________________________________________________________________

________________________________________________________________________

## 3.5 The Execution Time of an x86 Program

Chapter Three in the textbook provides the timings for each of the processors in the x86 family. By referring to these tables you can determine how many "clock cycles" the x86 processors will take to execute a program[8]. By going through and carefully counting the clock cycles, you can optimize the performance of your programs. Consider the program from the previous example:

```
            load  dx, 5       ;7-8 cycles
            load  cx, 0       ;7-8 cycles
            load  ax, 0       ;7-8 cycles
lp:         get   bx          ;1 cycle
            add   ax, bx      ;7 cycles
            add   cx, 1       ;9-10 cycles
            iflt  cx, dx, lp  ;9-10 cycles
            put   ax          ;1 cycle
            halt              ;0 cycles
```

To compute how long this code would take to execute on the 886, you must add up the timings for the instructions inside the loop and multiply this number by five (since the loop executes five times) and then add up the cycles for the instruction outside the loop[9]. For the code above that turns out to be somewhere between ((1+7+9+9)*5 + 22) cycles and ((1+7+10+10)*5 + 25) cycles; that is, somewhere between 152 and 165 cycles.

Simply claiming that the program will take between 152 and 165 cycles is somewhat nebulous. Now on some processors this is the best you can do. With the 886 processor we can figure out the timing *exactly.* The instructions which have a range of cycles times rather than a single value do not choose one of the values in the range at random. Consider the "LOAD reg,constant" instruction. If you look it up in the table you'll find that it takes between seven and eight cycles to execute. The reason this instruction doesn't always execute in the same amount of time is because the x86 processors are all 16-bit processors. If an instruction is three bytes long (like the LOAD reg,constant instruction) it contains a 16-bit constant following the opcode. If this constant falls on an even address the processor can fetch the constant in one clock cycle. However, if the constant begins at an odd memory address it will take two clock cycles to fetch the constant. This is why the 886 processor timing table lists these instructions as requiring between seven and eight cycles to execute. "LOAD reg, constant" requires seven cycles if the instruction begins at an *odd* address (which will put the constant at an even address). It takes eight cycles if the instruction begins at an even address.

Most of the other 886 instructions whose execution time varies by one clock cycle do so for the same exact reason.

The only exceptions are the instructions utilizing the [BX] addressing mode. These instructions are all one byte long so there isn't an extra cycle consumed by the processor when it fetches a 16-bit constant following the opcode at an odd address. There is no such constant following the opcode! However, this is a memory addressing mode which does read a word from memory (from the address contained in BX). If BX contains an odd value, the 886 will have to read a word from an odd address, taking an extra cycle. If BX contains an even value, the 886 processor can read all 16 bits in one clock cycle.

Those instructions whose timings vary by two clock cycles are all three byte instructions which reference memory. These guys can lose one clock cycle if the 16-bit constant following the opcode is at an odd address *or* if the memory location specified is at an odd address. These instructions will require *two additional clock cycles* if the 16-bit constant is at an odd address *and* the memory location specified is at an odd address.

8. These tables do no include timings for the GET, PUT, and HALT instructions. Mainly because these instructions take an indefinite amount of time to "execute". Simply use one cycle for GET and PUT and zero cycles for HALT in your computations.

9. Program analysis, that is, figuring out how long a piece of code takes to execute, is a very deep subject. This text will consider this topic a little later. But be ye forewarned– there are volumes written on this subject. This text will not give it a complete treatment.

3.35 Use the SIM886 **E**nter command.

The following table provides the exact timings for the current example:

| Address | Code | Instruction | Cycles |
|---|---|---|---|
| 0 | 1F | LOAD DX, 5 | 8, const is at odd adrs |
| 1 | 05 | | |
| 2 | 00 | | |
| 3 | 17 | LOAD CX, 0 | 7, const is at even adrs |
| 4 | 00 | | |
| 5 | 00 | | |
| 6 | 07 | LOAD AX, 0 | 8 const is at odd adrs |
| 7 | 00 | | |
| 8 | 00 | | |
| 9 | F1 | GET BX | 1 |
| A | 41 | ADD AX, BX | 7 |
| B | 57 | ADD CX, 1 | 9, const is at even adrs |
| C | 01 | | |
| D | 00 | | |
| E | B3 | IFLT CX, DX, 0009 | 10, const is at odd adrs |
| F | 09 | | |
| 10 | 00 | | |
| 11 | F8 | PUT AX | 1 |
| 12 | E0 | HALT | 0 |
| 13 | | | |

As written, this code will take exactly 159 clock cycles to execute. Note too bad, but it can be made faster by simply moving the instructions around. Consider the following code which does exactly the same thing:

```
                load  dx, 5
                load  cx, 0
                load  ax, 0
lp:             add   cx, 1
                get   bx
                add   ax, bx
                iflt  cx, dx, lp
                put   ax
                halt
```

**3.37 Provide the machine code for the above program *assembled beginning at address 0001***

________________________________________________________________

________________________________________________________________

________________________________________________________________

________________________________________________________________

**3.38 How many cycles (exactly) will the above code take to execute? ______**

You must be very careful when letting cycle counts control the way you write code. Cycle counting is something you should put off until the very end of the optimization cycle. Generally, there are lots of other optimizations you should do before relying on arcane techniques like cycle counting. Cycle counting tends to make your code harder to read and very inflexible. As an example, suppose you decide to add a single one-byte instruction to the beginning of this program. That would wipe out all the work you put into optimizing the code. Since programs tend to change quite a bit, optimization achieved by counting cycles and rearranging instructions, as above, is often wasted effort.

On the other hand, cycle counting is a technique available to assembly language programmers and should not be overlooked. Compilers use this technique to generate better code. You're putting yourself at a disadvantage if you're not aware of this optimization technique. For often-called library routines which are fully debugged, cycle counting can often produce dramatic results (especially on pipelined and superscalar CPUs).

There are a couple of optimizations we can yet perform to get the speed of this program down some more. The previous optimization consisted solely of *code motion*, that is, moving instructions around. You can also improve the execution time of this code segment by using *different instructions* to accomplish the same task. Consider the following code:

```
            load  cx, 0
            load  ax, cx
            load  dx, 5
lp:         get   bx
            add   cx, 1
            add   ax, bx
            iflt  cx, dx, lp
            put   ax
            halt
```

**3.39 Assemble this code *beginning at address 0001* and provide the machine code**

____________________________________________________________

____________________________________________________________

____________________________________________________________

____________________________________________________________

____________________________________________________________

**3.40 How many clock cycles (exactly) will this code take to execute?**

____________________________________________________________

## 3.6 Using the Built-in "Mini-Assembler"

The SIM886 program contains a built-in *miniassembler* module. This allows you to enter programs using x86 mnemonics rather than hexadecimal opcodes. This assembler does *not* provide many of the features you'd expect from a normal assembler (like MASM), but it is easier to use than the Enter command for enter x86 machine instructions.

To use the miniassembler, simply type "A" followed by the address where you wish the first instruction placed:

A *address*

SIM886 will prompt you to enter an x86 machine instruction. After you enter the instruction and hit the enter key, SIM886 will respond by printing the opcode and instruction, it will then expect you to enter a new assembly language instruction. You can continue entering instructions in this fashion until you've entered all your instructions. Hitting the enter key by itself on a line will terminate instruction entry.

Although the SIM886 Assemble command lets you enter instructions like "LOAD AX, 1234" it does not let you use symbolic labels for addresses and constants. In particular, you cannot use statement labels like the "lp:" label employed in the previous examples. You have to supply the hexadecimal address as the target of an IFxx or GOTO instruction. This isn't too hard if you're jumping backwards (as in the previous example), it is a problem if you're jumping forward and you haven't figured out the target address.

3.36 No, the IFLT instruction would transfer control to location 0009, not 1009 which is the new location of the GET instruction.

There are two ways to handle this problem. The first is to figure out the length of each instruction in your program. This is much easier to do than figuring out the opcodes since all instructions without a constant are one byte long, those with a constant are all three bytes long. Once you know the length of each instruction, you can figure out the address of each instruction (by summing the lengths of the previous instructions). If you write all this down on a sheet of paper before you enter your program, you'll know the target address of all jumps (IFxx and GOTO instructions) when you enter them.

A second alternative is to enter an arbitrary address (like zero) whenever you have a forward jump. Later on, when you enter the target instruction, you can go back and *patch* the jump to the target location. Consider the following code:

```
LBL1: IFLT  AX, BX, LBL2
      GET   AX
      GOTO  LBL1
LBL2:
```

When using the assemble command, you'll know the address to supply for the GOTO instruction (zero if you begin assembling this code at address zero). That's because SIMN886 prints the address of each instruction as it assembles it. However, when you enter the IFLT instruction, you really won't know the address associated with LBL2 until after you enter the GOTO instruction (assuming you haven't computed this manually beforehand). Suppose, however, that you entered the following:

3.37 1F 05 00, 17 00 00, 07 00 00, 57 01 00, F1, 41, B3 09 00, F8, E0

```
0:    IFLT  AX, BX, 0
3:    GET   AX
4:    GOTO  0
7: enter
```

After entering these instructions, it's clear that the target address for the IFLT instruction is seven. Although you've already entered the (incorrect) target address of zero, there is nothing stopping you from using the miniassembler to *replace* the IFLT instruction with the correct version. You can do this by typing "A *address_of_IFLT*" and then reenter the correct IFLT instruction, e.g.,

```
A 0
0: IFLT AX, BX, 7
3: enter
```

While the miniassembler isn't perfect, using it will speed your data entry considerably. Although this is a big step above hand assembling instructions yourself, just wait— a real assembler is vastly better yet.

## 3.7 Simple x86 Programming Techniques

While proficiency at x86 machine language programming isn't desirable, you will need to know a little bit about x86 programming in order to successfully complete the laboratory exercises. This section discusses some elementary programming techniques you can use when creating x86 machine language programs.

A variable's value is generally kept in a register or a memory location. Since the x86 hypothetical processors only have four registers, it is unlikely you will be able to keep all your variables in the registers for any non-trivial program. Therefore, you will need to keep most of your variables in memory locations and load them into CPU registers when you need to use their values. The x86 processors only provide 16 bit registers, therefore they operate on word values; hence, your memory variables will generally require two bytes each. To "create" a variable all you do is choose a pair of bytes in memory and use those two memory locations to hold the value of the variable. The only thing to remember is that each variable should have its own unique location in memory. If you use the same location for two separate variables, they will interfere with one another.

Keep in mind that the x86's memory space is an array of bytes. Therefore, each variable will require two consecutive bytes in memory. To initialize a variable with a value before your program runs, you can use SIM886's *enter* command to store a value into the memory locations you reserve for a variable.

Some of the laboratory exercises use *arrays* of data. An array is nothing more than a contiguous sequence of variables in memory. For example, an array containing five values would simply be ten consecutive bytes (one word for each of the five values in the array). To obtain the value of an array element, you would normally use the xxxx[bx] addressing mode. The xxxx displacement should be the address of the first element of the array. Then you can use the value in bx to select an element of the array. Since each element of the array is two bytes long, bx will need to contain even values to properly select an array element. For example, if you decide to store a ten element array starting at location 8000h in memory, the first element will be at address 8000h, the second will be at 8002h, the third will be at 8004h, etc. For example, suppose memory location 7000h contains an index (0..9) into the array of ten items starting at address 8000h and you want to load that particular array element into the ax register. You could use the following code to accomplish this:

```
            load  bx, [7000]        Get index into the array.
            add   bx, bx            Double bx's value.
            load  ax, 8000[bx]      Get desired array element.
```

Since all computations must take place inside the CPU, you must load any values you want to add or subtract into registers. For example, if you want to add the values at locations 1000 and 1002 together and then subtract the word at memory location 1010 from this sum and store the result at location 1008, you would use code like the following:

```
            load  ax, [1000]
            add   ax, [1002]
            sub   ax, [1010]
            store ax, [1008]
```

To make decisions or create loops in your programs, you will need to use the if*xx* and goto instructions. Using these instructions requires a little more work because you need to know the *target* address of the instruction they (may) jump to. For example, you may want to add one to memory location 800 if it is less than or equal the value in memory location 1000. In pseudo-code, this is fairly trivial:

```
if (memory[800] <= memory[1000]) then
        memory[800] := memory [800] + 1;
```

The standard way to translate this into assembly code is to jump *around* the code that increments memory location 800 if it *is not* less than or equal to the value at location 1000:

```
            load  ax, [800]
            load  bx, [1000]
            ifgt  ax, bx, ????
            add   ax, 1
            store ax, [800]
????:
```

Note that "greater than" is the opposite of "less than or equal."

The only problem is that we don't know the target address. This is especially problematic when jumping to code that has yet to be written (as is the case above when writing the ifgt instruction). The solution is to enter *any* value for the target address (most people use zero). Later, when you know the actual address of the target instruction, you can go back and reenter the ifgt instruction using the correct target address. For example, first you would enter the following:

```
0-          load  ax, [800]             Addresses supplied by SIM886's
3-          load  bx, [1000]            assemble command.
6-          ifgt  ax, bx, 0             Just use zero for now.
9-          add   ax, 1
C-          store ax, [800]
F-                                      Target address is 0F.
```

After entering this code, we know that the target address is 0Fh. Therefore, we can go back and reenter the ifgt instruction with the proper address. Since the ifgt instruction is at address six, you can use the SIM886 "A 6" command to do this:

3.38 153

```
A6
6-          ifgt  ax, bx, 0F
```

To continue adding additional instructions to your program, you can continue assembly by using an "A 0F" command.

Although the x86 processors only provide if instructions that test for less than, equal, or greater than, it is easy to test for less than or equal, greater than or equal, or not equal. The following code sequences demonstrate how to transfer to location 0F on these three conditions:

```
            iflt  ax, bx, 0F              test for <=
            ifeq  ax, bx, 0F
      < drop down here if greater than >

            ifgt  ax, bx, 0F              test for >=
            ifeq  ax, bx, 0F
      < drop down here if less than >

            ifgt  ax, bx, 0F              test for not
equal
            iflt  ax, bx, 0F
      < drop down here if equal >
```

3.39 17 00 00, 02, 1F 05 00, F1, 57 01 00, 41, B3 08 00, F8, E0

Loops are another common control structure you will need to use in your programs. The easiest loop construct to implement with x86 code is the *repeat..until* loop. This type of loop executes some code, tests and condition, and then repeats this process if the condition requires. If you need to execute a section of code a fixed number of times, the repeat..until construct is easiest to use. The following code example demonstrates how to store zeros into the five elements of an array starting at location 1000h:

```
0-          load  bx, 0            Starting array index
3-          load  dx, 10           Five iterations
6-          load  ax, 0            Zeros we will write
9-          store ax, 1000[bx]     Write a zero to the
array
C-          add   bx, 2            Move to next array
element
F-          iflt  bx, dx, 9        Repeat five times
12-         halt
```

3.40 150

Note that this loop compares bx with 10 on each loop iteration. This is because it adds two to bx since each array element is two bytes long. Therefore, the loop terminates after executing five times.

The second type of loop you will commonly use is the while loop. Unlike the repeat..until loop, the while loop tests the termination condition at the beginning of the loop, before it executes the body of the loop. The following sample program scans through the array at memory location 8000h searching for the first zero value, adding one to all non-zero values, stopping when it encounters a zero value. Note the use of the [bx] addressing mode vs. the xxxx[bx] addressing mode found in the previous example:

```
0-          load  ax, 0       The value to search for
3-          load  bx, 8000    Start of our array
6-          load  cx, [bx]    Get next array element
7-          ifeq  cx, ax, 16  Quit if this is a zero
A-          add   cx, 1       Add one to this value
D-          store cx, [bx]    Store back to memory.
10-         add   bx, 2       Move on to next element.
13-         goto  6           Repeat loop
16-         halt
```

For more information on encoding various high level language control constructs in an assembly language, see the chapter on control structures in the textbook. These simple examples should suffice for now; with them you should be able to easily code the various programming projects necessary for the laboratory exercises.

## 3.8 The SIM886 Laboratory Exercises

In this laboratory you will experiment with the SIM886 program. You will hand assemble, enter, and execute short 886 programs. You will observe the action of the program and note the execution times of the program. Finally, you will attempt to optimize your programs using code motion.

### 3.8.1 Before Coming to the Laboratory

Your pre-lab report should contain the following

- A copy of this lab guide with all the questions answered.
- A write-up on the SIM886 program explaining, in your own words, how to use each of the commands available in the debugger.
- A write-up on the 886 explaining, in your own words, how each of the instructions and addressing modes operate.

See Chapter Two of this laboratory manual for an example pre-lab report.

Note: your Teaching Assistant or Instructor may elect to give a quiz before the lab begins on the material covered in the laboratory. You will do quite well on that quiz if you've properly prepared for the lab and studied up on the stuff prior to attending the lab. If you simply copy the material from someone else you will do poorly on the quiz and you will probably not finish the lab. Do not take this pre-lab exercise lightly.

### 3.8.2 Questions for the Laboratory Exercises

**3.41 How do you turn on the printer to capture screen output?**

______________________________________________________________

**3.42 What SIM886 command could you use to send the screen output to a text file?**

______________________________________________________________

**3.43 What is the command that instructs SIM886 to read its input from a text file?**

______________________________________________________________

**3.44 What SIM886 command do you use to load and assemble the** LAB3_3.886 **text file?**

______________________________________________________________

### 3.8.3 Laboratory Exercises

In this laboratory you will perform the following activities:

- Demonstrate the use of the SIM886 debugger and all of the commands in the debugger
- Demonstrate the operation of all the 886 instructions and addressing modes.
- Enter several 886 machine language programs into the debugger and single step through the programs to execute them.
- Use the debugger to modify the operation of the programs.
- Explore a technique known as *self modifying code.*

❏ Exercise 1: The SIM886 debugger. Demonstrate the use of each of the debugger commands in the SIM886 program. Turn on the printer or capture screen output to a file to create a log of your work. Include this printout with your lab report. As usual, you should go through this exercise once before turning on the printer or screen capture, just to make sure you understand what you're supposed to be doing. Although you're not expected to produce a perfect printout (everyone is going to make a few mistakes) there is no need to include pages and pages of errors in your printout because you couldn't figure out what you were doing while the printer or screen capture was running.

To reduce the amount of time you spend in the laboratory, you should create a small text file containing all the commands you want to execute *before* coming to the laboratory. Then instruct SIM886 to read these commands from that text file rather than from the keyboard.

❏ Exercise 2: Machine code. The following is a short 886 program that inputs a value from the user and then reads that many integers from the user in a loop, produces the sum of those integers, and then prints the result. For example, if the user enters three, the program would prompt the user for three different numbers and then display the sum of those three numbers.

```
A 0
            get   cx
            load  bx, 0
            load  dx, bx
            ifeq  cx, dx,10
            get   ax
            add   bx, ax
            sub   cx, 1
            goto  5
            put   bx
            halt
```

Note: when actually executing this program, always use small values when specifying how many integers to enter. Otherwise the program will take too long to execute.

Encode this program using *machine code* and enter it into memory using the SIM886 *enter* command. Use the *unassemble* command to verify the correctness of your opcodes.

**For your lab report**: explain, in detail, how you generated each opcode for the instructions in this program. Capture your use of the **E**nter command while inputting the opcodes. Include this in your lab report. Run the program and capture its output (or send the output to a printer). Include this output in your lab report.

For the remaining programs in this laboratory, you can use SIM886's built-in miniassembler to input your programs.

**For additional credit**: Also input the program above using the SIM886 assemble command. Capture this process to a text file and include it with your lab report.

## 3.45 Provide the machine code for the following program:

| Address | Code | Instruction |
|---|---|---|
| 0 | | GET CX |
| 1 | | LOAD BX, 0 |
| 2 | | |
| 3 | | |
| 4 | | LOAD DX, BX |
| 5 | | IFEQ CX, DX, 10 |
| 6 | | |
| 7 | | |
| 8 | | GET AX |
| 9 | | ADD BX, AX |
| A | | SUB CX, 1 |
| B | | |
| C | | |
| D | | GOTO 5 |
| E | | |
| F | | |
| 10 | | PUT BX |
| 11 | | HALT |

❑ Exercise 3: Assembly code. The following short program reads two values from the user, a count and an address. It then writes *count* zero words to memory starting at *address*.

```
A 0
            get   cx          ;The count goes here
            get   bx          ;The address goes here
            load  dx, 0
            load  ax, dx
            ifeq  cx, dx, 13
            store ax, [bx]
            add   bx, 2       ;Each word in memory is two bytes!
            sub   cx, 1
            goto  6
            halt
```

This program is available on the diskette accompanying this manual (LAB3_3.886). Load this program into memory.

Step through the program one instruction at a time. Look at the memory locations pointed at by the BX register. Verify that this code is zeroing out sequential words in memory with each iteration of the loop.

Note that this code increments bx by two on each iteration of the loop. This is because each word consumes two bytes of memory and we need to point bx at a new word after each iteration of this loop.

**For your lab report:** Capture your session while you are running this program to a text file. Insert comments into the output (handwritten is okay if you dump the output to a printer) to describe what is going on during the execution of the code. Be sure to specify a small number of locations to zero (e.g., five).Suppose we had only added one to bx on each iteration of the loop. How many bytes would contain zero after executing this new version of the code given the two inputs 5 and 100? (hint: think this one through carefully.)

3.41 Press the control p keys to toggle the printer on and off.

**For additional credit**: modify the program so that you only add one to bx on each iteration of the loop. Run the program and capture the output. Be sure to dump the affected memory locations after each execution of the store instruction. Describe what is going on in your lab report.

❏ Exercise 4: Cycle counting. The following programs computes the sum of ten numbers starting at address 100h in memory. Using the execution times for the 886 processor given in your textbook, try to guess how long each instruction in this program will take to execute:

3.42 C *filename*

```
A 0
      load  cx, 20
      load  bx, 100
      load  ax, 0
      add   ax, 100[bx]
      add   bx, 2
      ifeq  bx, cx, 15
      goto  9
      put   ax
      halt
```

3.43 I *filename*

**For your lab report:** Run this program using the SIM886 **G**o command and the global cycle counter. Compare the actual cycle times SIM886 reports against the cycle times you predict. Be sure to describe any differences and discuss why your predictions were incorrect (unless you are among the 1% who got it right).

Note: this program appears on the diskette accompanying this lab manual (LAB4_3.886).

❏ Exercise 5: Effect of operand alignment. Repeat exercise four with the following slight modification to the above program:

```
A 0
      load  cx, 20
      load  bx, 100
      load  ax, 0
      load  ax, ax            ;Effectively a NOP
      add   ax, 100[bx]
      add   bx, 2
      ifeq  bx, cx, 16
      goto  a
      put   ax
      halt
```

**For your lab report**: Discuss why the timing for this program is quite a bit different from the one in exercise four.

Note: this program appears on the diskette accompanying this lab manual (LAB5_3.886).

❏ Exercise 6: Wait states. Repeat exercise five, except set the number of wait states to one. Repeat it again with the number of wait states set to two.

**For your lab report:** Discuss the effects of adding wait states to memory. In particular, discuss the performance loss between zero and one wait states and the performance loss (percentage) between one and two wait states. Which causes the greatest loss of performance (percentage)? Discuss how a cache might alleviate this problem.

**For additional credit:** Predict the effect of adding additional wait states to memory. Then try it and see how close you come to being right.

❏ Exercise 7: Self-modifying code. Von Neumann machines use the same memory to store instructions as data. This opens up the possibility that a program could actually rewrite itself on the fly. That is, the program could fetch some data, store it into memory somewhere, and then GOTO this data just stored into memory. Consider the following short instruction sequence:

```
0000:        load  ax, 001Fh
0003:        store ax, [000Ch]
0006:        load  ax, 0E000h
0009:        store ax, [000Fh]
000C:
000F:
```

The first four instructions store the bytes 1F, 00, 00, and E0 starting at location 000C (which just happens to be the first byte after the second STORE instruction above. If you disassemble these four bytes, you'll discover that they correspond to the instructions "LOAD DX, 0" and "HALT". So after this code segment executes the four instructions above it will load DX with zero and then stop.

To the beginning programmer self-modifying code looks really neat. After all, what could be more clever than a program which writes part of itself? Sounds like something out of Star Trek or some science fiction novel. In reality, there are many reasons to *avoid* using self-modifying code in your programs. First of all, most modern processors are sufficiently powerful that self-modifying code is not required.

Another way to use self-modifying code is to support *indirection*. Consider the following short code sequence:

```
2000: load  ax, [1000h]
        .
        .
        .
      load  bx, 2001h
      add   bx, 2
      store bx, 2001h
        .
        .
        .
      goto  2000
```

This short example fetches the word starting at location 1000h into AX and (presumably) performs some manipulations on that value. Later in the code it fetches the value from address 2001h. *This is the address of the 16-bit constant in the LOAD AX, 1000h instruction!* It adds two to this value and stores it back to location 2001h. This effectively changes the LOAD AX, [1000h] instruction to a LOAD AX, [1002h] instruction. Each time through the loop this program modifies the instruction at address 2000h to fetch a different value from memory.

As you've probably figured out already, there are easier ways of accomplishing this task (fetching successive words from memory) on the 886 processor; just use the [BX] or xxxx[BX] addressing modes. However, it's worth taking a look at self-modifying code for one very important reasons– many programs *accidentally* modify themselves. That is, bugs in the program cause it to store data over valid instructions whereupon the CPU executes those instructions with disastrous results. It is important that you realize that programs *can and will* modify themselves if you are not careful.

The following code is a slight modification to the program in exercise three. However, it uses self-modifying code rather than indirection to access memory. Enter this program (LAB7_3.886 on the diskette) and run it. Between each iteration of the loop, unassemble the program to see how this code modifies itself.

```
A 0
            get    cx            ;The count goes here
            get    [0c]          ;The address goes here
            load   dx, 0
            load   ax, dx
            ifeq   cx, dx, 1d
            store ax, [0]
            load   bx, [0c]
            add    bx, 2
            store bx, [0c]
            sub    cx, 1
            goto   6
            halt
```

3.44 F2
0F 00 00
19
93 10 00
F0
48
77 01 00
EF 05 00
F9
E0

❏ Exercise 8: Synthesizing other instructions. As you may have noticed, the instruction set of the x86 processor family is quite small – it only has 11 instructions. However, it is very easy to synthesize the operation of other types of instructions using these 11 instructions. For example, consider the following statements that compute the logical AND of ax and bx leaving the result in ax, assuming ax and bx contain zero or one:

3.45 I LAB3_3.886
3.46

```
1000:       ifeq  ax, bx, 1006
1003:       load  ax, 0
1006:
```

If ax is equal to bx, then they are both one or they are both zero. In either case, the value in ax is the result of ax AND bx. If they are not equal to one another, then one of the two registers contains zero so the result must be zero. The code above handles this task.

Another important pair of instructions missing from the x86 instruction set is the subroutine *call* and *return* instructions. It is easy to transfer control to a subroutine using the GOTO instruction, the only problem is how does the subroutine know where to return control when it completes execution? The trick is to have the calling routine supply a *return address* when it calls the subroutine. The subroutine then jumps to this return address when it wants to return to the caller. Consider the following code sequence:

```
100:        load  ax, 109     ;109 is the return address.
103:        store ax, [1000]  ;Save return address here.
106:        goto  1002        ;Routine starts at 1002.
109:                          ;Subroutine returns here.
             .
             .
             .
1002:       load  dx, 0             ;Compute NOT BX
1005:       ifeq  bx, dx, 100E      ;See if BX is 0 or 1
1008:       load  bx, 0             ;If bx=1, set it to 0
100B:       goto  [1000]            ;Return to caller
100E:       load  bx, 1             ;If bx=0, set it to 1
1011:       goto  [1000]            ;Return to caller
```

Assemble the code above and trace through it to verify proper operation.

**For your lab report**: Discuss how the subroutine call and return operations work. Run the program and include a screen capture with your lab report.

**For additional credit:** Write a subroutine that computes the logical OR of the cx and dx registers. Run your code, capture the output, and include the screen capture with your lab report.

❏ Exercise 9: Memory-mapped I/O. In addition to the GET and PUT instructions, the x86 processors support *memory mapped I/O.* Memory mapped I/O systems use memory locations as the interface to external devices. SIM886 provides support for 12 memory-mapped I/O devices: the eight LEDs and four switches on the circuit you built for Chapter Two's laboratory. Memory locations 0FFE0h through 0FFEFh correspond to the eight LEDs. SIM886 writes the L.O. bit of each of these eight words (FFE0/FFF1, FFE2/FFE3, FFE4/FFE5, etc.) to the corresponding LEDs. These are read/write locations. You can read any value you write to these locations. Memory locations FFF8/FFF9, FFFA/FFFB, FFFC/FFFD, and FFFE/FFFF are *read-only* locations. Any value you write to these memory locations will be lost. These memory locations will contain zero or one depending upon the status of the corresponding switch. If you read a one, then the corresponding switch is in the on position. If you read a zero, then the switch is in the off position. Consider the following program:

```
A 0
            load  ax, [fff8]        ;Read SW1
            store ax, [ffe0]        ;Write value to LED0
            add   ax, 1             ;Tricky way to invert L.O. bit.
            store ax, [ffe8]        ;Write inverted value to LED4
            load  ax, [fffa]        ;Read SW2
            store ax, [ffe2]        ;Write value to LED1
            add   ax, 1
            store ax, [ffea]        ;Write inverted value to LED5
            load  ax, [fffc]        ;Read SW3
            store ax, [ffe4]        ;Write value to LED2
            add   ax, 1
            store ax, [ffec]        ;Write inverted value to LED6
            load  ax, [fffe]        ;Read SW4
            store ax, [ffe6]        ;Write value to LED3e
            add   ax, 1
            store ax, [ffee]        ;Write inverted value to LED7
            goto  0                 ;Repeat forever.
```

The program above will continually read the switches and write their values and their inverted values to the LEDs until you press Ctrl C on the keyboard.

**For your lab report:** Enter and run the code above. Describe the operation of this program in your lab report.

**For additional credit:** After reading the switch inputs, compute some logic functions based on those inputs and write the function results to the LEDs.

## 3.9 Programming Projects

- ❏ Program #1: The x86 processors do not have a multiply instruction. Obviously it is sometimes necessary to multiply two values together. One way to perform a multiplication is to add a number to itself some number of times. Given two integers in locations 1000 and 1002, write a short code sequence which multiplies these two integers together and prints their product then terminates. Implement this code inside SIM886 using the miniassembler. Trace through the program after initializing location 1000 with 3 and 1002 with 4. Produce a print-out of the trace. Try running the program at full speed (using the "GO" command) with larger values in locations 1000 and 1002.
- ❏ Program #2: Modify the above program so that it reads the two integers from the keyboard before multiplying them.
- ❏ Program #3: The x86 processors do not have a division instruction. However, you can simulate division in software using repeated subtractions (and counting the number of times you can subtract the divisor from the numerator). Design an x86 division algorithm. Trace through the algorithm for small values. Use the GO command when dividing large numbers by small numbers. Hint: the IFLT and IFGT instructions deal with *unsigned* numbers only. You can simulate a test for less than zero by checking to see if one value is *greater than* 7fffh. For the division algorithm you will want to subtract the divisor from the numerator until the numerator is less than zero. The number of subtractions you've performed, minus one, is the quotient.
- ❏ Program #4: Write a short program which inputs two numbers, doubles the first value the number of times specified by the second, and prints the result. Note: to multiply a value by two, load it into a register and add that register to itself. Create a loop to add the register to itself the number of times specified by the second input value.
- ❏ Program #5: Write a short program that will compute the sum of *n* numbers in memory, where the user supplies the starting address and the number of entries to add together.
- ❏ Program #6: 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. Use the subroutine call and return mechanism described in the laboratory exercises.
- ❏ Program #7: Write a subroutine that computes the logical OR of the values (zero or one) in the AX and BX registers. Call this subroutine three or four times from your main program with different values in AX and BX. Use the subroutine call and return mechanism described in the laboratory exercises.
- ❏ Program #8: Write a subroutine that extracts a bit from a 16-bit number. The subroutine should expect the number in the AX register and it should return the bit position of the highest-order set bit in the number. Return 0FFFFh if AX contains zero.
- ❏ Program #9: Write the generic logic function for the x86 processor. 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 32,767.
- ❏ Program #10: Write a program that reads the generic function number for a four-input function from the user and then continually reads the switches (using the memory-mapped I/O scheme, see the laboratory exercises) and writes the result to the LEDs.

## 3.10 Answers to Selected Questions

1. CPU, Memory, I/O.

6a. 8 6f. 32

7a. 20 7f. 32

10/286 1 2 1 2 1 2

16. The clock frequency is the reciprocal of the clock period.

20a. 1 20c. 2

21. There are other delays in the system due to buffers, latches, and decoding which sit between the CPU and memory.

26. Generally the CPU reads several consecutive words (a cache *line*) into the cache at one time. This saves having to fetch groups of consecutive bytes from memory on each access.

28a. Four out of five accesses require 0 wait states (total 1 cycle each). One out of five accesses requires a total of 11 clock cycles (10 wait states). Total for five accesses is 15 clocks producing an average of 2 wait states per memory access (3 clocks per memory access minus the one clock which the CPU normally requires).

33. The prefetch queue allows the CPU to overlap fetching one instruction with the execution of the previous instruction. Consider the two instruction sequence "`add ax, bx`" and "`store ax,[1000h]`". While the CPU executes the "`add ax, bx`" operation, the busses are free and the CPU can, in parallel, fetch the opcode bytes for the "`store ax,[1000h]`" instruction.

40. If you have separate instruction and data caches the CPU can fetch instruction opcodes and data operands during the same clock cycle, thereby executing these operations in parallel. Without the cache, the CPU would have to execute separate memory cycles for each fetch.

44. AL, BL, CL, DL, AH, BH, CH, DH

52. Carry, Overflow, Zero, and Sign flags are the condition codes.

## Machine Language Coding Form #1:

| Address | Code | Instruction | Cycles |
|---|---|---|---|
| 0 | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| A | | | |
| B | | | |
| C | | | |
| D | | | |
| E | | | |
| F | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |
| 13 | | | |
| 14 | | | |
| 15 | | | |
| 16 | | | |
| 17 | | | |
| 18 | | | |
| 19 | | | |
| 1A | | | |
| 1B | | | |
| 1C | | | |

## Machine Language Coding Form #2:

| Address | Code | Instruction | Cycles |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |