USING THE ASSEMBLER

It is important to understand how instructions are assembled and stored in memory, but entering any but the smallest programs by hand is very tedious. For most of the programming you will be doing in the rest of COCO, you will want to use the **AS11** assembler program (or **ASM.BAT**) instead.

Creating Source Code Files

To create an assembly program for the 68HC11, use your favorite PC text editor or word processor. If using a word processor, be sure to save your source code in **text only** or **ASCII** format. The formatting in any word processor file format will really confuse the assembler! The standard suffix for assembly listings is **.ASM** or **.asm**. Use this suffix to designate all of your source code files.

Writing assembly code is analogous to the way you write source code in most other languages, such as C. There is one difference from what you may be used to, however. Our assembler (and many others) assumes fixed fields in the source file. Anything starting in the first column is assumed to be a label; the first word not in column one is assumed to be an instruction mnemonic or assembler directive. The third field contains the operands, if any, and the fourth contains comments and is ignored by the assembler. All fields are separated by one or more spaces or tab characters (**white space**). Note that this means that you cannot put spaces in the operand field! Whatever is after the space will then be interpreted as a comment. Another important thing to remember is that the assembler only resolves symbol and label names to eight characters. You can make them longer if you want, but be careful. The names **VARIABLE1** and **VARIABLE2** will be treated as identical by the assembler. For readability, please align all your instructions.

Assembling Programs

When you are finished typing in the program, you must assemble it into machine instructions with the assembler. The lab PCs are equipped with the assembler program **AS11**, which creates 68HC11 machine code in S-record format from assembly listings. For more information about the format of S-records, refer to the EVB manual (HC11EVB.pdf). You do not need to invoke the assembler itself; instead, use the batch file **ASM.BAT** that was created especially for this class. To assemble the program, type **ASM <filename>** at the **DOS** prompt, where **<filename>** is the name of the file that contains the source code, minus the **.ASM** suffix. For example to assemble the program called **MYPROG.ASM**, you would type **ASM MYPROG** at the **DOS** prompt. The batch file will invoke the assembler and several files will be created. These files can be identified by their suffix as follows.

.**ASM**	The original source listing.
.**LST**	The assembler output listing.
.**SYM**	A file that defines a program's labels for use by the simulator.
.**S19**	The machine instructions in S-record format.

The assembler output is a detailed listing that shows you exactly what machine instructions were created, and where they will be placed in memory, along with the original source code. The assembler output listing also contains the error messages that were generated during assembly. If your code did not assemble correctly, an **.S19** file will not be generated (or it will be empty). Consult the **.LST** file to find out what the errors were.

If you wish to simulate your source code, make sure that a **.SYM** file was generated. This file contains information that the simulator uses to map memory locations to labels that you have used in your code. This makes the disassembly more readable and recognizable to you. It also allows you to use symbolic labels to reference points in your code. For example, within the simulator, you may type **BR MYLABEL** rather than **BR <hex_address>** to set a breakpoint within your code. The **BUFFALO** monitor has no capability to recognize labels.

Assembling an Example Program

Now you are ready to try the simulator on a more complex example. For this assignment, you will need to enter the code from the first programming example into the simulator. To do this, type the code on the next page into a text file, then assemble it using the **ASM** batch file. If the assembler generates any errors, examine the **.LST** file, correct your mistakes and re-assemble the code. After assembling the code with no errors, run the simulator, and load the assembled program (the **.S19** file) into the simulator with the **SCRIPT <filename>** command. When the program is loaded, trace through the code and write down the appropriate register values in the blanks provided below.

REGISTER and MEMORY VALUES
(first time through the program only)

Label	Operation and Operand	A (hex)	X (hex)	CCR (ls 4 bits)	TEMP (hex)
	`ORG $C000`				
`TEMP`	`RMB 1`				
`START`	`LDX #TABLE`				
	`CLR TEMP`				
`LOOP`	`CPX #ENDTABLE`				
	`BEQ DONE`				
	`LDAA 0,X`				
	`INX`				
	`CMPA TEMP`				
	`BLE LOOP`				
	`STAA TEMP`				
	`JMP LOOP`				
`DONE`	`SWI`				
`TABLE`	`FCB 5, 2, 23, 25, 10, 50, 100, 57, 250, 200`				
`ENDTABLE`					

Question: When the program ends, what is the final value of **TEMP**? ________________

The ASCII Code

The ASCII chart is on the last page of your 68HC11 *Programming Reference Guide* pocket handbook. Lookup the ASCII codes for the following.

Symbol	**ASCII code (in hex)**
Lower case “a”	
Upper case “A”	
Control-C	
The number “9”	
The number “5”	
The “#” symbol	

WRITING YOUR OWN ASSEMBLY LANGUAGE PROGRAM

By now you should be ready to write a program of your own. It will be an application of the **exclusive-OR** logic operation. Store a string of **ASCII** bytes (call the string *message*), the last byte of which is **0**, and a single byte called *key* in the memory. Then, write a simple loop program that performs the **exclusive-o**r of the *key* with each byte of the *message* string. Overwrite the characters of the original message with the results. When it finds the **0** string terminator, stop. The following C-like program can represent this simple algorithm.

```
char key;

char message[] = “Secret Message!”;

/* C strings are automatically terminated by \0 , an ASCII 0 */

int  x = 0;

while ( message[x] != ‘\0’ )

{

message[x] = message[x] XOR key;

x++;

}

exit(0);
```

Debug and run the above program on the simulator. Figure out which byte is the **KEY** (the .**LST** file has this information), and modify that memory location using the **MM** command. Give the **KEY** the value **$AA**. Next, figure out which memory location your program code starts in. Run the program once, then look at the message string. It should be quite unintelligible. Next run the program again. The message should be back to normal! This simple scheme works because the **XOR** operation is completely reversible. What it does to each data byte is to flip every bit (0→1, 1→0) in the message where the corresponding bit of the key is a 1. (The key and data bytes are viewed as binary numbers.) When the program is run again with the same encryption key, all the flipped bits just get flipped back.

For fun, you can try running the program with other keys. The keys $00 and $20 should prove interesting. Can you figure out why they do what they do?