

# Programs

N. Narasimhamurthi

## 1 Objective

To become familiar with writing programs. This lab also illustrates the use of assembler directives.

### 1.1 What you should do

You will have to turn in your LST files for the programs you write for this lab. Check with the TA for additional instructions.

### 1.2 Programs

When you write programs in a high level language such as C, C++, Java, the compiler facilitates modular program development using function. In an earlier lab, we discussed some basic rules for writing functions. Another facility that higher level languages provides is the use of variables. Key features of variables in high level languages are:

1. Variables generally have a name and you use the name to manipulate them.
2. Variables have a type. The compiler keeps track of the type and makes sure that the way a variable is used conforms to its type.
3. Compiler negotiates with the operating system to obtain adequate memory for the variable.
4. Compiler makes sure that a memory allocated to a variable is not accidentally also allocated to another variable
5. Compiler keeps track of the life and scope of the variable so that the variable is available only when it is in the scope of the instruction (the simplest case is the distinction between global and local variables.

In assembly code, you are pretty much on your own. First of all, variables are known by their address. Some variables require more than one location. In this case, you allocate consecutive memory locations for the variable, and the address of the variable is the first of these locations. It is your responsibility to make sure that (a) you set aside adequate memory for the variable, and (b) you do not assign the same location for more than one variable. The most common errors that programmers make is not keeping this in mind. For example, the programmer may set the address of a two byte variable as \$3120 and the address of another variable as \$3121. **It is a bad idea for you to manually assign addresses to variables.** Let the assembler do it for you. However, if you have to manually assign an address to a variable, EQUate a label to the variable as

```
TOTAL EQU $3800 *variable to keep track of the total
```

The above instruction defines a variable called TOTAL that is stored in locations starting from \$3800. Note you have no idea how many bytes are needed for the variable and therein lies the potential bug!

### 1.2.1 How the assembler works

Inside the assembler is a variable called the *location counter*, also known as the *dot* in the unix community. When the assembler starts, the location counter is initialized to zero. As the assembler reads your program, the location counter changes in response to your code. For example, the ORG command sets the location counter to the value after the ORG instruction. Thus, if you want to change the location counter to \$00EB, you would write

```
ORG $00EB
```

The assembler keeps a copy of the HC11 memory internally, and the location counter is used to address the memory. Here, briefly, is how the assembler works:

1. Read the instruction. If the instruction has a label, and it is not an EQU instruction, then equate the label to the location counter. If the instruction is an EQU instruction, the label is EQUated to value after the mnemonic.
2. If the instruction is a valid HC11 instruction, generate the code for the instruction. The code is stored in consecutive memory locations starting from the address given by the location counter. The location counter is incremented by the amount of memory needed to store code.
3. If the instruction is a RMB instruction, then add the value after the mnemonic to the location counter.

4. If the instruction is a FCB command, then a sequence of comma-separated values following mnemonic are stored in consecutive locations starting from the address given in the location counter. The location counter is incremented by the amount of memory needed to store these bytes. FCB is a convenient way to specify a sequence of ASCII characters. Thus the following two statements are equivalent

```
FCB 72, 69, 76, 76, 79
FCB /HELLO/
```

The FDB instruction is similar to FCB, except the values are interpreted as 16-bit numbers. *These initializations are done in your PC and then transferred to the HC11 via the S19 file. If these memory locations in your HC11 are modified after the transfer, either by your program or by accident, then you have to reload your S19 file! Also, DO NOT use these commands to initialize variables. Your program will work only once. If you rerun your program, the variables will not be reinitialized!*

Here is an example of defining variables and constants (variables that your program will not modify).

```
ORG $3000 ;Start of data section.
V1      RMB 4 ; set aside 4 bytes. EQUate V1 to first address
V2      RMB 11; set aside 11 bytes
THOU    FCB 3, $E8 ; Initialize 16-BIT variable called THOU
BUFF    FILL $22,18 ; same as fcb with $22 repeated 18 times
BUF2    RMB 20
OPT s ; turn on symbol dump option
```

It is a good idea to turn on the symbol dump option. This will cause assembler to print all the symbols at the end of your program listing. Type in the above sequence of instructions and assemble it. Look at the LST file and write down what the symbols V1, V2 etc., are EQUated to and explain the results.

### 1.3 Your first program

We will now write the first program. When writing assembly code, a convenient way to document your code is to write the *pseudocode*. Rather than invent another pseudo language, I will use a C like syntax<sup>1</sup>. The program we want to write should be similar to the following C program (since HC11 is essentially an 8-bit micro, all variables will be unsigned chars to keep things simple).

---

<sup>1</sup>This reverses history! C language was invented to avoid writing assembly code!

```

#include <stdio.h>
unsigned char v1, v2, v3, v4; unsigned char
total; main() {
    v1 = 11;
    v2 = 0x2F; /* In C prefix 0x denotes HEX */
    v3 = 'A';
    v4 = 044; /* In C prefix 0 denotes octal */
    total = v1+v2+v3+v4;
    printf("%02X", (unsigned) total);
    return 0;
}

```

Compile and run the above program using any C compiler. The output of your program should be 9F. Now assemble the following HC11 program, and run it to verify that you get the same answer:

```

;Name: etc
;
; This program adds 4 numbers and prints the answer.
;
outlhlf    equ $ffb2
outrhlf    equ $ffb5

    org $2000
main
    ; v1 = 11;
    ldaa #11
    staa v1

    ;v2 = 0x2f;
    ldaa #$2f
    staa v2

    ;v3 = 'A';
    ldaa #'A'
    staa v3

    ;v4 = 044;
    ldaa #@44
    staa v4

```

```

;total = v1+v2+v3+v4;
ldaa v1
adda v2
adda v3
adda v4
staa total

;print total as 2 digit hex number
ldaa total
jsr outlhlf
ldaa total
jsr outrhlf

swi

org $3000
v1 rmb 1
v2 rmb 1
v3 rmb 1
v4 rmb 1
total rmb 1

opt s

```

Modify the C program so that `v3 = 'A'`; is replaced by `v3=getchar()`; . Run the C program and type the upper case letter A and press enter. You should see the same output as before. Now replace the instruction `ldaa #'A'` by `jsr inchar` and make sure that you equate `inchar` to `$ffcd`. Run the assembly above assembly language program and verify that it behaves the same as the C program.

## 1.4 On your own!

1. Write an assembly program that does the same as the following C program:

```

#include <stdio.h>

char v;
char total;
main() {
    v = 0;
    total = 0;

```

```

v = v+1;
total = total + v;

v = v+1;
total = total + v;

v = v+1;
total = total + v;

v = v+1;
total = total + v;

v = v+1;
total = total + v;

printf("%02X", (unsigned) total);
return 0;
}

```

2. Modify the program so that the initialization `v = 0` is replaced by a call to `getchar()`. Run your C program and try different inputs at the keyboard. Make corresponding changes to the assembly language program and verify your results.
3. Modify the assembly language program so that you use a counting loop to loop 5 times over the basic code.
4. Modify the previous version (using a loop) where the number of times around the loop is in an 8-bit *variable* called `count`. Your program should initialize the variable to 5 so that the loop is executed 5 times.
5. A convenient way to get a value between 0 and 9 from the user is to use the instruction sequence

```

jsr inchar
anda #$0F
; in C, use: getchar() & 0X0F

```

Use the above sequence to let the user specify the count. When expecting the user to enter a value, it is a good idea to prompt the user. Your code may look something like:...

```

;; various equates and comments...
;
;
    org $2000
    ldx #prompt
    jsr outstrg
    jsr inchar
    anda #$0f
    staa count
    ;
    ; rest of the code goes here

    org $3000
    ; various RMB
    count RMB 1
    v    RMB 1
    total RMB 1

prompt fcc /How many times please? /
       fcb 4

```

6. Write a program that will do the following:

- (a) Get a number between 0 and 9 from the user and store in a variable called `v1`.
- (b) Multiply `v1` by 4 and store the result in a variable called `v2`.
- (c) Add `v1` and `v2` and store the result in a variable called `v3`.
- (d) Multiply `v3` by 2 and store the result in a variable called `v4`.
- (e) Print the values in the variables separated by a comma. In the last lab, you saw an example for printing a colon. You can use the same approach.
- (f) Modify the program so that the output reads some thing like:

```

v1 = 07
v2 = 1C
v3 = 23
v4 = 46

```

7. Write a function that will be passed a value in **A** register. The function should return 10 times the value passed to it. Use the above function to write a program that does the following: The program should get a digit from the user, multiply it by 10 and store the value in a variable called **tens**. It should then get a digit from the user and add it to the variable **tens** and store it in a variable called **DecimalIn**. The program should then print the value in the variable.
8. Use the code you wrote to create a function that will read a two digit decimal number from the user. Call this function **ReadDecimal**. Write a program that will call this function to read a two digit number and store the value in a variable called **DecimalIn**. The program should then print the value in the variable.
9. Type the following C program, run it to see what the output is. Rewrite the program in assembly language and run it on the HC11. Verify that output of the assembly program matches the C program.

```
#include <stdio.h>

char delta, value;
char count;
main() {
    value = 0;
    delta = 1;
    count = 11;

foo:
    if (count == 0) goto bar;

    printf("%02X", (unsigned) value);
    putchar(','); putchar(' ');

    value = value + delta;
    delta = delta + 2;
    count = count -1;
    goto foo;

bar:
    return 0;

}
```

10. Type the following C program, run it to see what the output is. Rewrite the program in assembly language and run it on the HC11. Verify that output of the assembly program matches the C program.

```
/*  
  
    I decided to use fprintf(stderr instead of printf(  
    as printf( and getche don't mix well.  
  
    I am forced to use getche instead of getchar because  
    getchar puts the terminal in the 'cooked' mode rather  
    than 'raw' mode and there seems to be no way to uncook  
    the input in MS Windows.  
  
    The program as written will work on all variants of Windows  
    and on unix boxes if you know the right curses!  
  
    Note: getche is an exact equivalent of INCHAR in BUFFALO  
          To print a string, you use OUTSTRG in BUFFALO  
  
*/  
  
#include <stdio.h>  
#include <conio.h>  
  
char c;  
  
main() {  
foo:  
    fprintf(stderr, "\n\nWelcome! Your choices:\n\n");  
    fprintf(stderr, "\n1. Set temperature");  
    fprintf(stderr, "\n2. Set Speed\n");  
    fprintf(stderr, "\nChoice please: ");  
    c = getche();  
  
    if (c == '1') goto one;  
    if (c == '2') goto two;  
    fprintf(stderr, "\n\nNot a valid choice!\n");  
    goto foo;  
  
one:
```

```
        fprintf(stderr,"Good choice. \n");
        goto more;
two:
        fprintf(stderr,"Try later\n");

more:
        fprintf(stderr,"Try again? ");
        c = getche();
        if (c == 'y') goto foo;
        if (c == 'Y') goto foo;
        if (c == 'n') goto bye;
        if (c == 'N') goto bye;
        goto more;

bye:
        return;
}
```