Benjamin Linam

Bml42@students.uwf.edu

EEL 4744L: Microprocessor Applications Laboratory

Lab 5: Writing Subroutines and Using BUFFALO I/O Routines

2/28/2018

Objective

Introduce students to writing subroutines in HC11 assembly language and using the BUFFALO I/O routines to display results.

Introduction/Background/Theory

A subroutine is a set of instructions that can be used by a program several times throughout its execution, similar to functions used by other higher-level programming languages such as C. This lab required construction of an assembly program that includes subroutines which when called, display the elements in an NxM matrix in row-major order, as well as swap each of the rows of the array. The point of the lab is to construct each set of instructions to where it may be used several times to output the desired result.

To complete this program, certain BUFFALO monitor I/O subroutines must be used to display information to the console for the user to view. While the subroutines have set addresses in memory, their names are not recognized by the HC11; therefore, variables must be created that refer to their locations. After creating the variable, the desired subroutine can be called upon by using the command JSR to indicate that program execution must "jump" to the desired address of that subroutine.

Jumping between subroutines and the main body of the program is done by use of the stack pointer. The return address for each subroutine is saved to the stack for the program to be able to return to once the subroutine has completed its execution. Not only can addresses be pushed to the stack, but variables and other values can also be saved for easier access.

Procedure

1. As shown in Fig.1, program execution begins at $B600 with a message that displays that the original matrix will follow before calling the subroutine OUTSTRG to display the elements stored in the matrix.

```
        ORG     $B600           ;Save code in EEPROM
**** Start of Main Program ****
Main    LDS     #$01FF          ;Initialize SP

        LDX     #MSG1           ;Load X with base address of MSG1
        JSR     OUTSTRG         ;Call subroutine to print MSG1
        LDX     Matrix          ;Load address starting addr of Matrix to X

        BSR     PRINTMAT        ;Call subroutine to print original matrix

        BSR     SWAPMAT         ;Call subroutine to swap matrix columns

        LDX     #MSG2           ;Load X with base address of MSG2
        JSR     OUTSTRG         ;Call subroutine to print MSG2

        BSR     PRINTMAT        ;Call subroutine to print modified matrix
        SWI                     ;return to Buffalo monitor
```

Figure 1: Assembly language code showcasing the general layout of the main program which calls subroutines PRINTMAT and SWAPMAT

2. Fig.2 shows that PRINTMAT is then branched to and by incrementing through each element in the matrix as well as calling the OUTLHLF and OUTRHLF subroutines, each value is displayed to the console.

```
PRINTMAT    LDAA    #1      ;A points to the first element
            STAA    i       ;Store i in accumulator [A]
            CLR     j       ;Clear value stored in j
            LDAB    #M      ;stores accumulator [B] with M dimension
            LDX     #Matrix ;Load [X] with address of Matrix
            JSR     OUTCRLF ;outputs ASCII carriage and outputs the characters

Loop        LDAA    0,X
            JSR     OUTLHLF ;converts left nibble of A to ascii
            LDAA    0,X
            JSR     OUTRHLF ;converts right nibble of A to ascii
            LDAA    #$20
            JSR     OUTA    ;outputs space between
            LDAB    i       ;Load accumulator [B] with i
            CMPB    #M
            BEQ     NEXTEL

            BRA     NEXTROW ;Branch
NEXTEL      CLR     i       ;Clear value stored in i
            INC     j       ;Increment j
            JSR     OUTCRLF

NEXTROW     LDAB    j       ;load accumulator [B] with j
            CMPB    #N      ;Compare [B] with N dimension
            BEQ     LEAVE   ;Last row output to console, return from subroutine
            INC     i       ;increment i
            INX             ;increment [X] to point to next element
            BRA     Loop    ;Branch back to the loop
LEAVE       RTS             ;Return to subroutine
```

Figure 2: Assembly language code for PRINTMAT subroutine

3. After the matrix is displayed and the return is reached at the end of PRINTMAT, program execution resumes in the main portion of the function. The next instruction is for the program to branch to the SWAPMAT subroutine, shown in Fig.3.

```
SWAPMAT     LDAA    #0      ;Load accumulator [A] with 0
            STAA    i       ;Store [A] to i
            STAA    j       ;Store [A] to j
            STAA    TEMP1
            STAA    TEMP3
            LDAA    #M      ;Load accumulator [A] with M
            STAA    TEMP2
            LDAA    #N-1
            STAA    TEMP4
            LDAA    i
            LDAB    #M      ;Load accumulator [B] with M
            MUL             ;Multiply i and M
            ADCA    #0
            ADDD    #MATRIX ;Add accumulator D with MATRIX
            XGDX            ;Exchange [D] with [X]

LOOP2       LDAA    TEMP4
            LDAB    TEMP2
            MUL             ;Multiply TEMP4 and TEMP2
            ADCA    #0
            ADDD    #MATRIX
            XGDY            ;Exchange [D] with [Y]

LOOP1       LDAA    0,X
            LDAB    0,Y
            STAA    0,Y
            STAB    0,X
            INX             ;Increment X
            INY             ;Increment Y
            INC     TEMP1   ;Increment TEMP1
            LDAA    TEMP1
            CMPA    #M      ;Compare accumulator [A] with M
            BNE     LOOP1
            CLR     TEMP1    ;Clear TEMP1
            INC     TEMP3    ;Increment TEMP3
            DEC     TEMP4    ;Decrement TEMP4
            LDAA    TEMP4
            CMPA    TEMP3    ;Compare TEMP4 and TEMP3
            BGE     LOOP2   ;Branch if greater or equal

            RTS             ;Return to subroutine
```

Figure 3: Assembly language code for SWAPMAT subroutine

4. The subroutine has full access to the matrix by incrementing through each of its elements, and by use of temporary variables, it swaps each element to its desired location. Elements in the first row are swapped with the last row, second row elements are saved to the second to last row, etc. If there are an odd number of rows, the middle row remains untouched. Once the middle of the matrix is reached and there are no remaining swaps to be made, the subroutine returns to the main program.

5. OUTSTRG is used again to display a message stating that the modified matrix will follow it. Then the OUTCLRF subroutine moves the cursor to the next line and PRINTMAT is called for the last time to display the values stored in the matrix.

6. After having compiled the .s19 file and uploading it to the HC11, typing “g b600” in the BUFFALO monitor will display the following represented by Fig.4 and Fig.5.

7. The two sets of original matrix values were provided during the lab and had to be manually entered in the .asm file before compiling and uploading it each time.

```
BUFFALO 3.4 (ext) - Bit User Fast Friendly Aid to Logical Operation
>g b600

The original matrix is as follows:
01 02 03 04
05 06 07 08
09 0A 0B 0C

The modified matrix is as follows:
09 0A 0B 0C
05 06 07 08
01 02 03 04
```

Figure 4: BUFFALO I/O console displaying the 3x4 array before and after the swap

```
>g b600

The original matrix is as follows:
01 02 03 04 05
06 07 08 09 0A
0B 0C 0D 0E 0F
10 11 12 13 14

The modified matrix is as follows:
10 11 12 13 14
0B 0C 0D 0E 0F
06 07 08 09 0A
01 02 03 04 05
```

Figure 5: BUFFALO I/O console displaying the 4x5 array before and after the swap

Conclusions

The program was by far more difficult to complete than the previous two; however, it was demonstrated successfully without error. If this program were to have been written in the C language, it would have been much easier, but with assembly being a lower-level language, the loops had to be entered manually as well as many of the other instructions that C greatly simplifies. Being restricted to using the registers for almost every operation was another point of concern that in C, would have been much more simple.