Experiment No. 02:

TUTOR COMMAND UTILIZATION and PROGRAM EXPERIMENTATION

By: Christopher Udeh

Lab Partner: Hieu Nguyen

Instructor: Dr. Jafar Saniie

ECE 441-001

Lab Date: 09-06-2018

Due Date: 09-13-2018

Acknowledgment: I acknowledge all of the work (including figures and codes) belongs to me and/or persons who are referenced.

Signature: _____

I. Introduction

A. Purpose

The purpose of this experiment was to become more familiar with the SANPER-1 Unit as well as some basic M68K assembly programs.

B. Background

In Lab 1, we experimented with the basic functionalities of the SANPER-1 and the TUTOT software it operates on, in this lab we will be working on more advanced features which the SANPER-1 possesses, while also typing and modifying some MC6800 programs which are supported by the SANPER.

II. Lab Procedure and Equipment List

A. Equipment

- SANPER-1 system
- PC with TUTOR software

B. Procedure

PART A

- Type in the sample program 2.1 using TUTOR
- Initialize the register values of A0 and A1 to the values required by the program i.e. the start and ending address of the memory to be filled.
- Initialize the value of register D0 to the data that the memory is to be filled with.
- Use the BF command to initialize the memory region to be filled to a known value.
- Run the program using the G command.
- Use the MD command to confirm that the program ran smoothly.
- Rerun in trace mode and record the output.
- Modify the program to run using pre-decrement while performing the same task.
- Repeat the steps listed above and confirm the program still works correctly.

PART B

- Type in the sample program 2.2 using TUTOR
- Replace “??” in the sample program text with the ASCII code for the letter “A”
- Run the program using the "G" command.
- Confirm that the program runs as expected, and that A is repeatedly printed out to the terminal.
- Press the Abort button on the SANPER-1 and use the MM command and modify the ASCII code for “A” to the number 5.
- Run the program again and confirm that “5” is repeatedly printed to the terminal
- Modify 0xFFFF to 0x000F in the instruction located at address 0x90E.
- Run the program again and note anything different that occurs.

PART C

- Type in the sample program 2.3 using TUTOR.
- Replace the address values with 0x1000 and 0x1018.
- Use the MS command to store the string “WELCOME TO THE 68000 LAB” at location 0x1000.
- Use the BS command and confirm that the string was properly stored.
- Use the G command and run the program, confirm that the program ran correctly by making sure the stored string was printed to the terminal.

PART D

- Type in the sample program 2.4 using TUTOR.
- Store the string, “MC68000 MICROPROCESSOR” at address 0x2001 and 0x3001.
- Use the BS command and confirm that the string was properly stored.
- Load 0x16, the string length into address 0x2000and 0x3000.
- Initialize the result location 0x1100 to 0xAAAA.
- Use the G command and run the program, then verify the output at 0x1100.
- Alter to string to “MC6800 MICROPROCESSOR”, and repeat the steps above.

PART E

- Type in the sample program 2.5 using TUTOR.
- Enter a list of word size values starting at address 0x2200.
- Initialize the values of registers A0 and A1 to the beginning and end of the list.
- Use the G command and run the program, then verify the list has been correctly sorted.

PART F

- Store the new value to be inserted at register D0.
- Initialize the values of registers A0 and A1 to the beginning and end of the list.
- Use the G command and run the program, then verify the list has been correctly sorted, including the newly entered value.
- Modify the program to read a value entered by the user.
- Use the G command and run the program, then verify the list has been correctly sorted, including the value entered from the terminal.

III. Discussion and Analysis

Program 2.1

1. **Address Instruction Comments**

00300C MOVE.W D0, (A0) +; Move a word from D0 to memory pointed by A0, then increment A0
00300E CMP.W A0, A1; Compare the values stored in A0 and A1
003010 BGE $300C; If A1 >= A0, continue at the beginning of the program
003012 MOVE.B #228, D7; Move trap #14 code #228 to register D7
003016 TRAP #14; Call trap routine

2. ***Program 2.1 modified**

Address Instruction Comments

```
MOVE.W D0, -(A1);     Decrement A1 then move a word from D0 to mem. pointed to by the
new A1
CMP.W A0, A1;         Compare the values stored in A0 and A1
BGT $300C;            If A1 >= A0, continue at the beginning of the program
MOVE.B #228, D7;      Move trap #14 code #228 to register D7
TRAP #14;             Call trap routine
```

3. The functions of the registers are as follows:
   a. A0= Store starting address,
   b. A1= Store ending address,
   c. D0= Value to fill memory with.

4. The pre-decrement and post-increment address modes allow for fast and convenient access to data which is stored adjacent to each other, such as arrays. This is so because the increment and decrement actions can be performed simultaneously with the operation been performed.

Program 2.2

1. **Address Instruction Comments**

```
000900 MOVE.B #$41, D0;       Move the ASCII character value for A into D0
000904 MOVE.B #248, D7;       Output single character
000908 TRAP #14;              Call trap routine.
00090A MOVE.L #$FFFF, D5;     Move the hex value $FFFF into register D5
000910 DBEQ D5, $910;         If not zero AND D5 is still >= 0, branch back.
000914 BRA $900;              Go back to the beginning of the program
```

2. After performing step 10 in the lab manual, the output was printed to the screen faster, since less DBNE instructions were executed in the main loop as a result of changing $FFFF which is 65535 in decimal to $000F which is 15 in decimal.

3. A single character can be outputted using the following:

Address Instruction Comments

```
MOVE.B D1,D0;     Move byte to be printed
MOVE.B #248,D7;   Output single character
TRAP #14;         Call the trap routine
```

4. Changing the instruction at 0x914 to branch to 0x904 would cause wrong output since calling the TRAP #14 handler would replace the contents of D0.

5. The instructions form a busy-wait loop using the decrement and BEQ instruction to slow the iteration of the main loop.

6. The TRAP instructions is useful because it allows a user to call upon a predefined routine with supervisor privileges for a program running in user mode.

Program 2.3

1. **Address Instruction Comments**

```
MM $950; DI
000950 MOVE.L #$1000, A5;      Starting address of string buffer
000956 MOVE.L #$1018, A6;      Ending address of string buffer
00095C MOVE.B #227, D7;        Output a string + space to the terminal
000960 TRAP #14;               Call trap routine
000962 MOVE.B #228, D7;        Print the Prompt
000966 TRAP #14;               Call trap routine
```

2. If there were no TRAP function, the function would have to be implemented as a subroutine that iterated the length of the string, printing out each character individually.

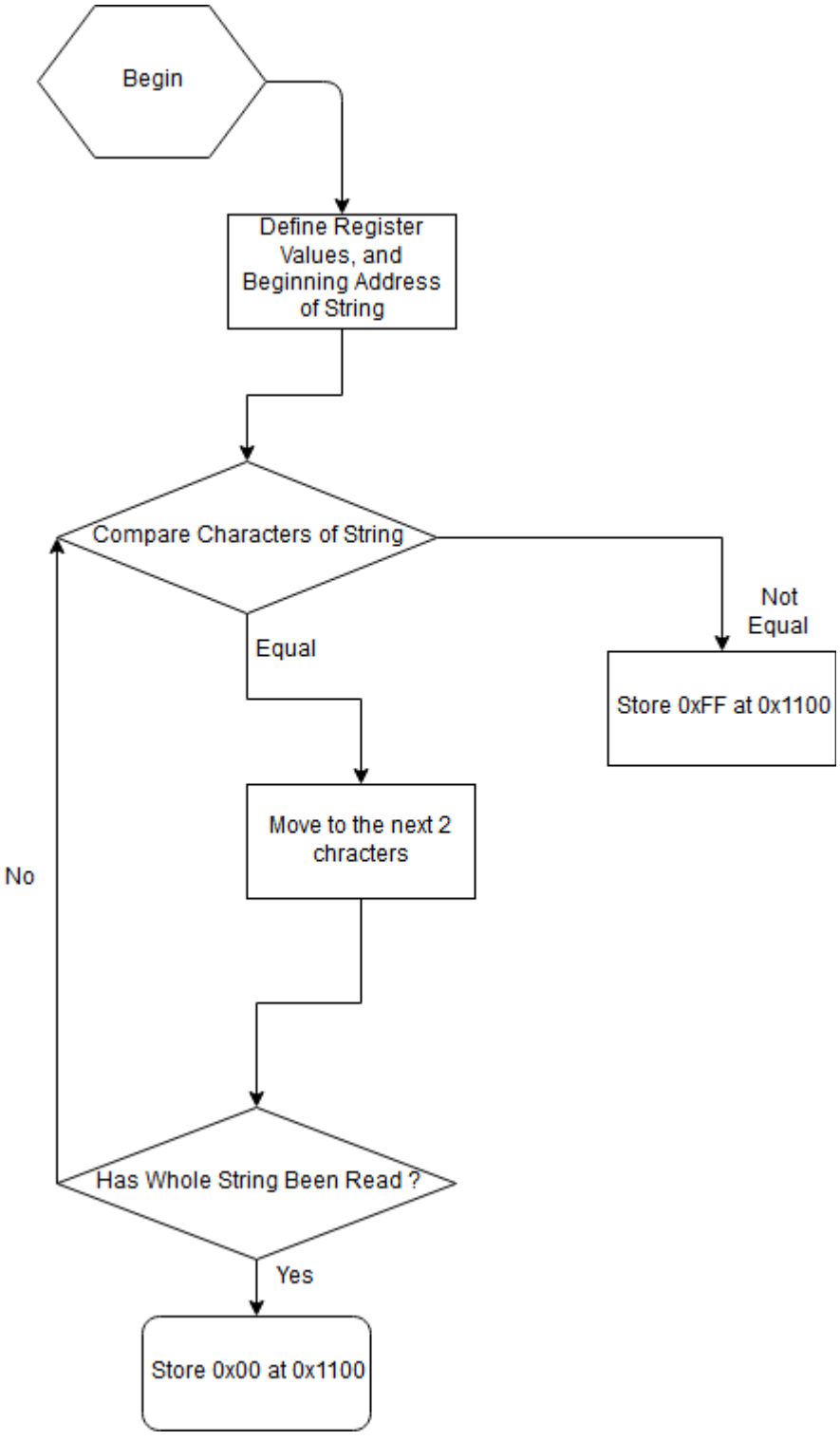
3. The value of register A5 after running the program is 0x1018, this is because according to the TRAP 14 documentation, the final value should be the ending address of the buffer + 1.

Program 2.4

1. **Address Instruction Comments**

```
MM $1000; DI
001000 MOVE.L #$2000, A0;       Start address of the first string
001006 MOVE.L #$3000, A1;       Start address of the second string
00100C MOVEQ.L #-1, D1;         Default result is 1
00101E MOVEQ.L #0, D0;          Clear D0
001010 MOVE.B (A0), D0;         Move length of the first string to D0
001012 CMPM.B (A0) +, (A1) +;   Compare the two immediate blocks of data
001014 DBNE D0, $1012;          Length is not equal -> end program;
001018 BNE.S $101C;             Branch to $101C if not equal
00101A NOT.B D1;                Flip the result
00101C MOVE.B D1, $1100;        Store result at memory $1100
001020 MOVE.B #228, D7;         Print the Prompt
001024 TRAP #14                 Call trap routine
```

2.


Begin
Define Register Values, and Beginning Address of String
Compare Characters of String
Not Equal
Store 0xFF at 0x1100
Equal
Move to the next 2 chracters
No
Has Whole String Been Read ?
Yes
Store 0x00 at 0x1100

3. The main difference between MOVE and MOVEQ is that for MOVEQ, the source operand of MOVEQ must be an immediate value but it executes faster than MOVE.

4. The CMPM instruction allows for comparisons with both source operands specified using post-increment addressing mode, which is extremely useful when comparing adjacent memory blocks such as arrays and strings.

5. The condition codes are set by the DBNE instruction.

Program 2.5

1. **Address Instruction Comments**

```
002000 MOVE.L A0, A2;         Make a copy of the starting address to A2
002002 MOVE.L A2, A0;         Restore the beginning index to A0
002004 CMP.W (A0) +, (A0) +; Compare contents of A0 with the next memory
002006 BHI.S $2014;            if n > n+1 branch to swap data
002008 SUBQ.L #2, A0;         Next contiguous pair
00200A CMP.L A0, A1;          Check for the end of the data table
00200C BNE $2004;             If not at the end of the list, sort the next pair
00200E MOVE.B #228, D7;     TUTOR prints prompt, end of program
002012 TRAP #14
002014 MOVE.L –(A0), D0;   Get two 16-bit number at once as a long to D0
002016 SWAP.W D0;             Swap the word at bits [15,8] with the word at bit [7,0]
002018 MOVE.L D0, (A0);      Put the swapped 16-bit numbers back to memory with the sorted
order.
00201A BRA $2002; Check the next pair of numbers
```

2. The sorting algorithm works by comparing the words in the list in pairs, if the pair of words is not properly sorted, the data is swapped. Every time a swap occurs the iteration begins from the top of the list again, when the program traverses the list without swapping anything, then the sort is complete.

3. The SWAP instruction is useful for swapping the upper word of a data register with the lower word of the register, it does not work on address registers however, only data registers.
Without the SWAP instruction, these instructions could be used:

```
MOVE.L D0,D1
LSL.L #16,D0
LSR.L #16,D1
OR.L D1,D0
```

4. The ADDQ and SUBQ instructions are faster than the regular ADD and SUB instructions, but are only valid when the source operand is an immediate value.

5. The main difference between ADD and ADDQ is that ADDQ only works when the source operand is an immediate value, while ADD has multiple addressing modes.

6. Like their ADD and ADDQ counterparts, the main difference between SUB and SUBQ is that SUB has multiple addressing modes while SUBQ only works when the source operand is an immediate value.

7. The instruction results in a comparison between the word pointed to by A0, as well as the next word, after this A0 then points to the next word that comes after the second word used in the current comparison.

***Program 2.6**

1. *D0 contains data to be inserted
   *A0: address of the first number
   *A1: address of the last number
   *Table sorted with higher numbers in lower memory addresses

Address Instruction Comments

003000 CMP.W (A0), D0; Compare number at current index with the number to be inserted
003002 BCC $300C; If the number is bigger than the current value in memory.
003004 MOVE.W (A0), -(A0); Shift data one word up
003006 ADDQ #4, A0; Move cursor to the next 2 words
003008 CMPA.L A0, A1; Check if at the end of list
00300A BCC $3000; If not the end of the list keep traversing the list
00300C MOVE.W D0, -(A0); Store the new number to the memory address above the current address
00300E MOVE.B #228, D7; Print the Prompt
003012 TRAP #14

*2.6 program added

Address Instruction Comments

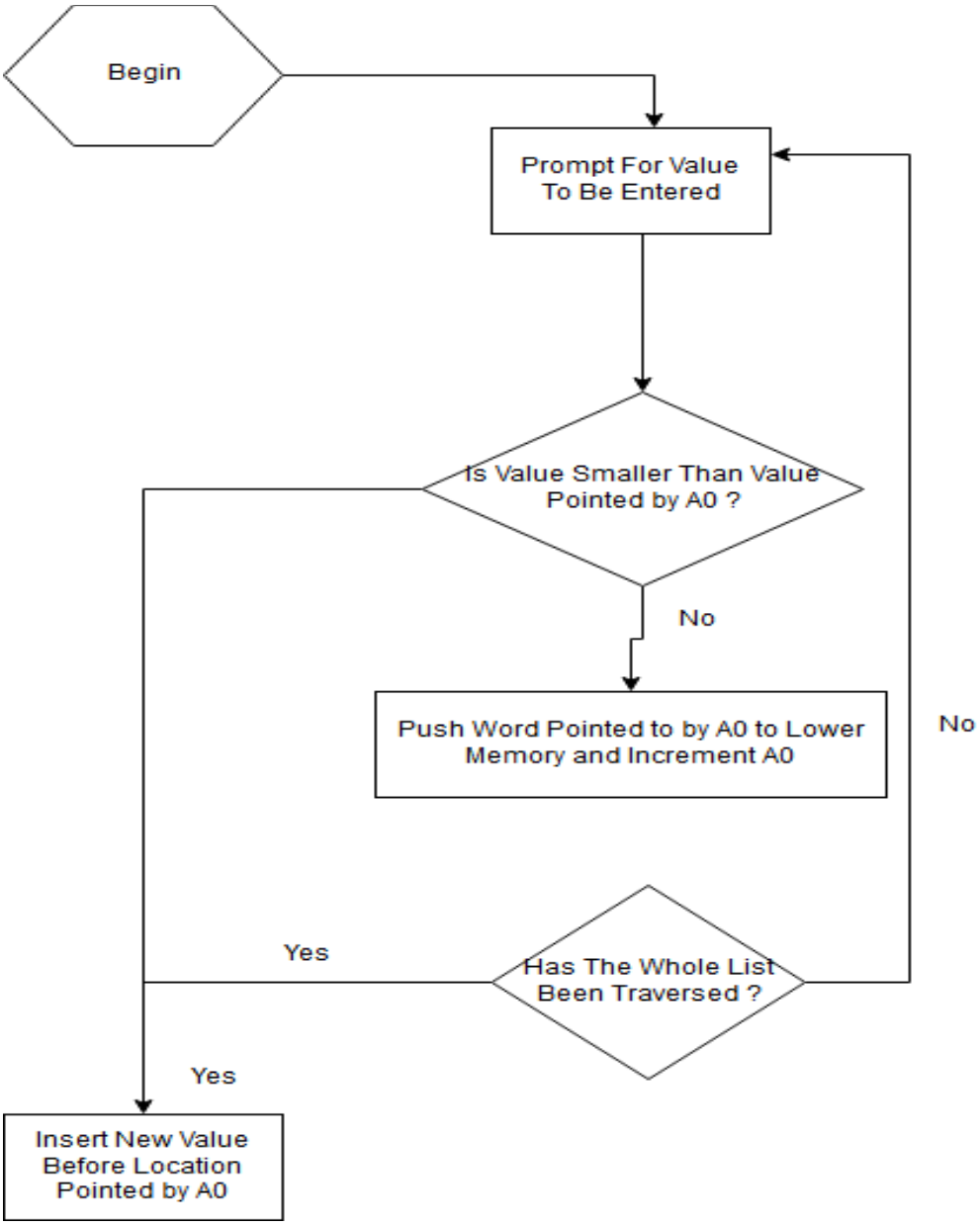
MOVE.L #$1000, A5; Start address of input buffer
MOVE.L #$1000, A6; Will be the end address + 1 of the input buffers
MOVE.B #241, D7; Enter a string from the terminal
TRAP #14
MOVE.B #226, D7; Convert ASCII encoded number to hex.
TRAP #14; Number is now stored in D0
RTS; Return to subroutine

*Call get input subroutine
MOVE.B #0, D7; Put the user defined program to be #0 in the TRAP 14 table
TRAP #14

*Register user defined function
MOVE.B #253, D7; Put the user defined program to be #0 in the TRAP 14 table

```
MOVE.L #StartAddressOfTheSubroutine, A0;    Put the user defined program to be #0 in the
TRAP 14 table
TRAP #14
```

2. The insertion algorithm loops over the array before each iteration to test if the correct position of the new value has been located. If the correct location is not found, the current words are pushed towards lower memory, which opens a space for the new value to be inserted in. This process goes on until the proper position is found or the end of the list is reached.



References:

SANPER-1 Manual

ECE 441 Lab Manual