ECE 441

Microprocessors

Instructor: Dr. Jafar Saniie

Teaching Assistant: Guojun Yang

Final Project Report:

**MONITOR PROJECT**

04/24/17

By: Javier Sorribes

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

Signature : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Table of Contents

1-) Abstract 2

2-) Monitor Program 2

2.1-) Command Interpreter 3

2.1.1-) Algorithm and Flowchart 3

2.1.2-) 68000 Assembly Code 5

2.2-) Debugger Commands 5

2.2.1-) Debugger Command #1: HELP 6

2.2.2-) Debugger Command #2: MDSP 7

2.2.3-) Debugger Command #3: SORTW 7

2.2.4-) Debugger Command #4: MM 8

2.2.5-) Debugger Command #5: MS 8

2.2.6-) Debugger Command #6: BF 9

2.2.7-) Debugger Command #7: BMOV 10

2.2.8-) Debugger Command #8: BTST 10

2.2.9-) Debugger Command #9: BSCH 11

2.2.10-) Debugger Command #10: GO 11

2.2.11-) Debugger Command #11: DF 12

2.2.12-) Debugger Command #12: EXIT 12

2.2.12-) Debugger Command #13: BPRINT 12

2.2.12-) Debugger Command #14: CONV 12

2.3-)Exception Handlers 13

2.3.1-) Exception Handlers Algorithm and Flowchart 13

2.3.2-) Exception Handlers Assembly Code 13

2.4-)Quick User Instruction Manual 16

3-) Discussion 17

4-) Feature Suggestions 18

5-) Conclusion 19

6-) References 20

7-) Appendix 21

1-) Abstract

The monitor program allows a user to enter commands and interact with a Motorola MC68000 processor and its memory.

In this report, all available commands are described, along with their implementations. Their corresponding algorithms, flowcharts and assembly codes are provided.

A similar description of the exception handling subroutines follows.

In addition, a quick user manual with the command usage can be found.

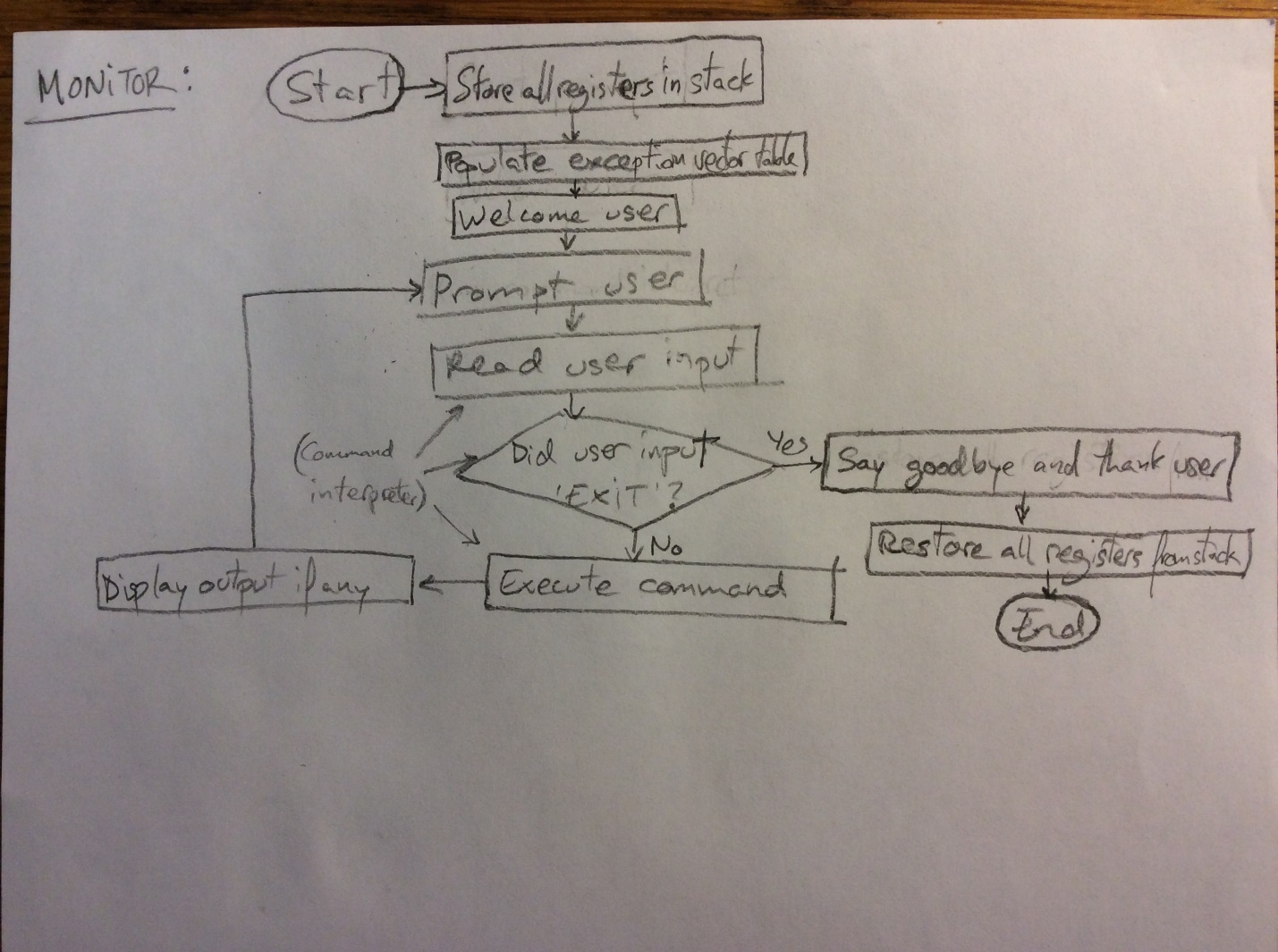
Furthermore, a discussion about challenges and uses of this project is proposed. This section is continued with feature suggestions for newer, more advanced versions of this program. Some conclusions about the project itself are also given.

Finally, external references and an Appendix with all of the code in the monitor program are provided at the end of the report.

By reading this report, the user will understand the usage and implementation of the monitor program, as well as the design and production process.

***2-) Monitor Program***

This program allows the user to enter an executable command into the console, sometimes providing the appropriate arguments. Then, the command is run, the output (if any) displayed. Finally, the prompt will be redisplayed and the process will start over. The user may run the ‘EXIT’ command to terminate the program. The following flowchart represents this process:



*Figure 2.1. Monitor Program Flowchart*

Descriptions of all the available commands, also named debugger commands for their hardware debugging capabilities, are outlined in the following sections.

In addition, this program accounts for asynchronous exceptions, providing exception handling routines. Their descriptions can also be found in the sections after those dedicated to the debugger commands.

Finally, note that many of the subroutines outlined in this report use other helper subroutines. These are not explicitly explained in this report, but refer to the Appendix for the code in these subroutines, which are simple enough to be understood from the assembly code and comments.

***2.1-) Command Interpreter***

The command interpreter compares the first word of the input against a table with all command names. These command names are preceded with a digit determining their length, which is used by the algorithm to know how to advance to the next row, or name. Each command name is also followed by either a null or space character, depending on whether the command takes arguments or not.

Once the command interpreter finds the command name in the table, it uses the offset within that table to access the correct memory location of the executable in a command location table. If the name is not found, then an invalid message is displayed and the program prompts again.

Note that the command interpreter does not parse the arguments of the command, but rather leaves that task to each command. This design decision was taken because each command may require a variable number of commands in different formats.

***2.1.1-) Algorithm and Flowchart***

An algorithm of the design and its flowchart are displayed below:

*COMMAND INTERPRETER*

*While input != ‘EXIT’*

*Print prompt*

*Read input into the stack*

*counter = 0*

*row 🡨 first row in command names table // row is name with length preceding*

*While row < last row in table*

*counter2 🡨 length of name from row*

*While counter2 > 0*

*If next byte of input == next byte in row // keep comparing*

*counter2 = counter2 – 1*

*Else // name is different from input*

*counter = counter + 1*

*row 🡨 next row*

*Break while loop*

*End while*

*If counter > # command names // name not in table*

*Print invalid message*

*Break while loop*

*Else if counter = 0 // name was found*

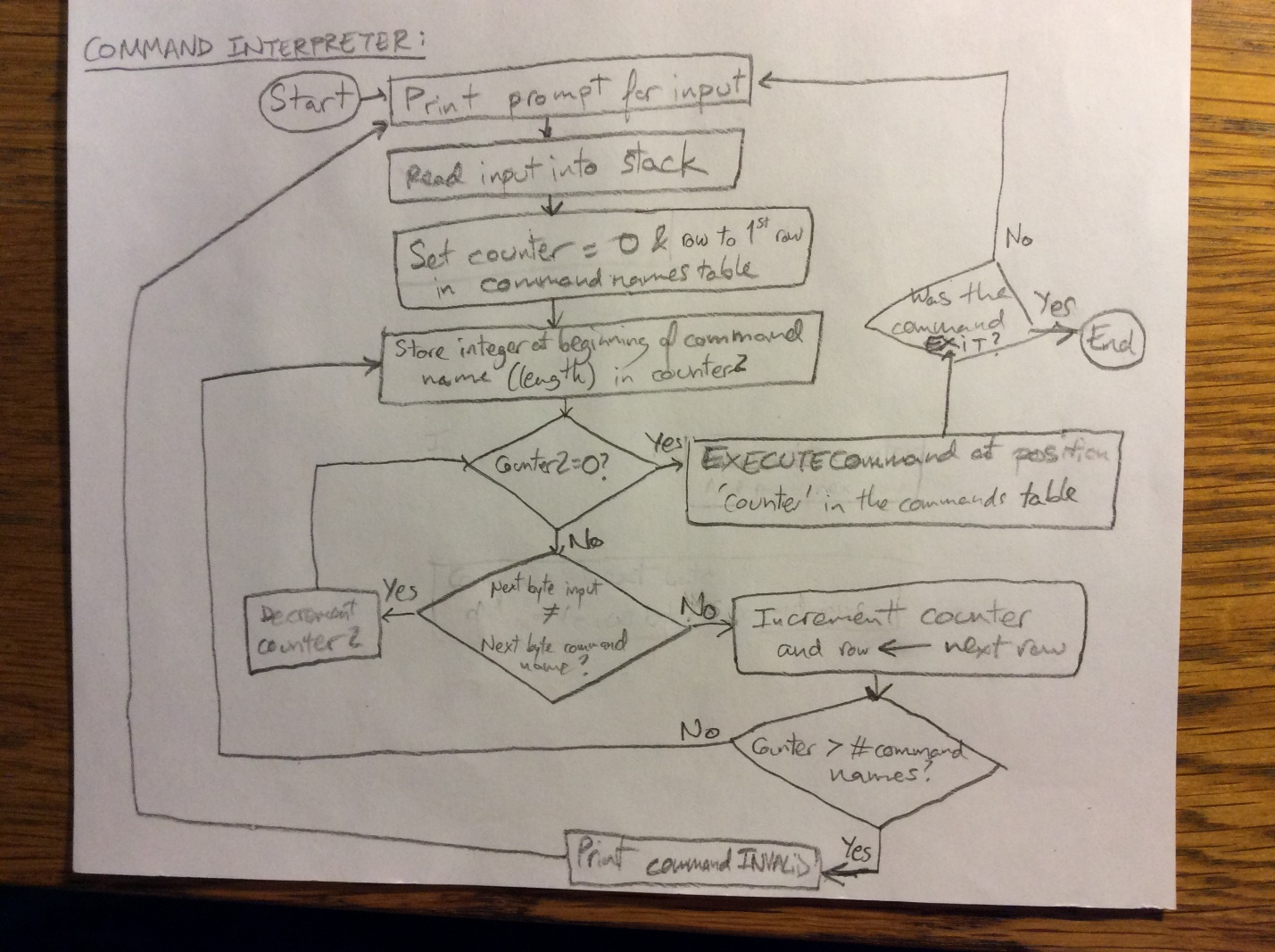
*Execute command at offset counter from command addresses table*

*End while*

*End while*

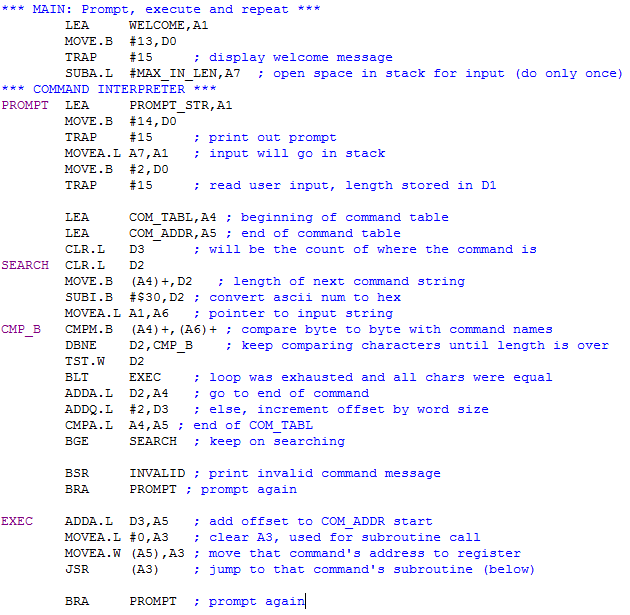
*Finish // finish*

*Figure 2.2. Command Interpreter Algorithm*



*Figure 2.2. Command Interpreter Flowchart*

***2.1.2-) Command Interpreter Assembly Code***



*Figure 2.3. Main & Command Interpreter 68000 Assembly Code*

***2.2-) Debugger Commands***

All debugger command subroutines store all used registers in the stack at the beginning and restore them at the end to ensure that nothing is overwritten. They each parse the arguments passed if anything, and display an invalid message if the usage is wrong. Then, they proceed to execute the corresponding algorithm and display any relevant output. Finally, they return to the main subroutine.

***2.2.1-) Debugger Command #1: HELP***

Displays the commands’ descriptions and usage. Prints the message in two parts to avoid not showing a part of it if it is too long.

***2.2.1.1-) Debugger Command #1 Algorithm and Flowchart***

*HELP*

*Print first part of message*

*While no input // wait*

*End while*

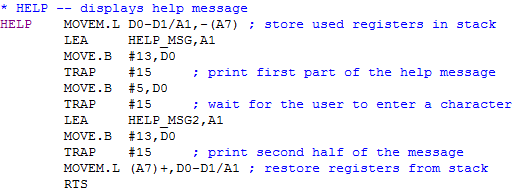
*Print second part of help message*

*Finish*

*Figure 2.4. Debugger Command # 1 Algorithm*

*Figure 2.5. Debugger Command # 1 Flowchart*

***2.2.1.2-) Debugger Command #1 Assembly Code***

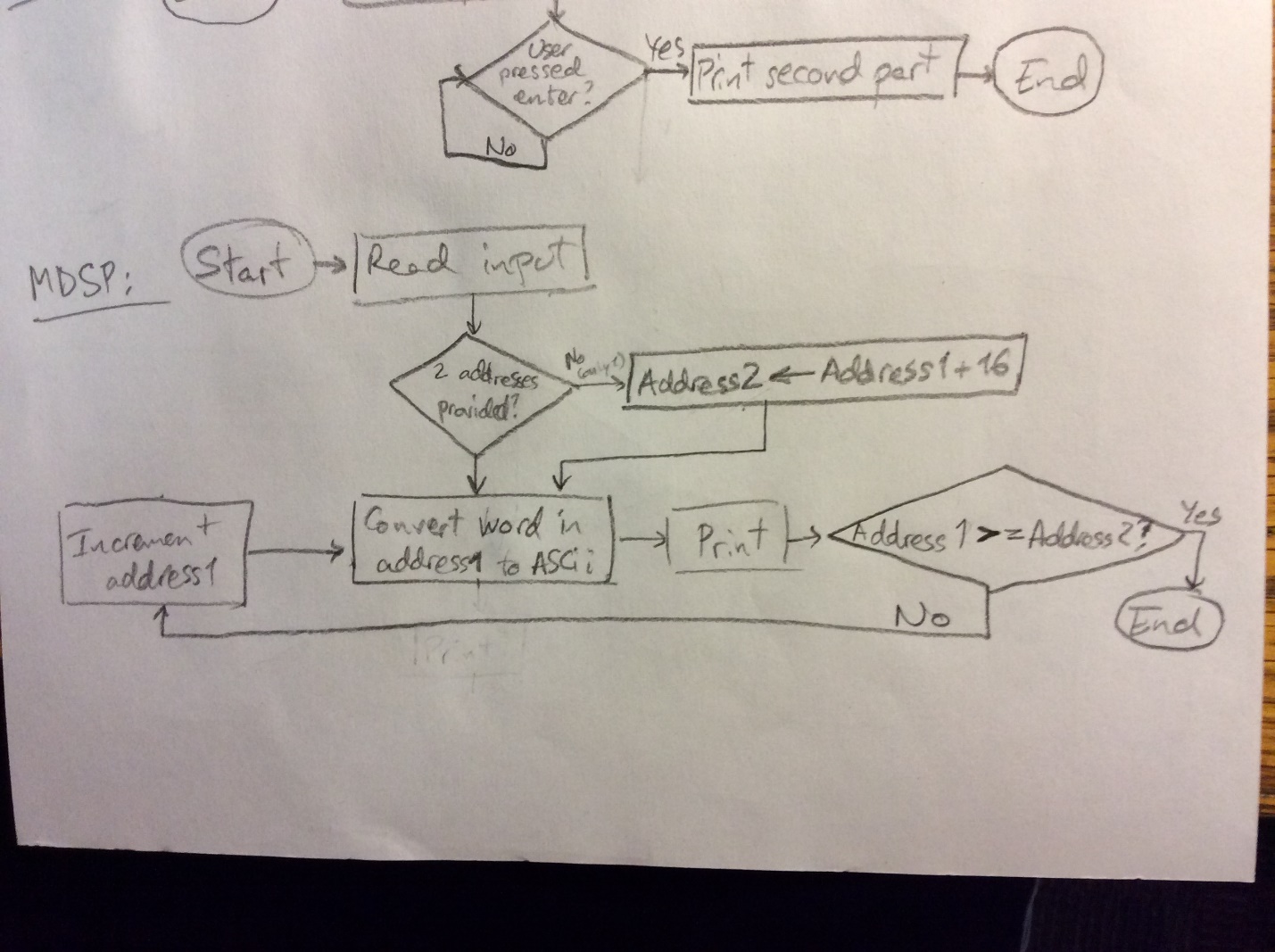


*Figure 2.6. Debugger Command #1 Assembly Code*

***2.2.2-) Debugger Command #2: MDSP***

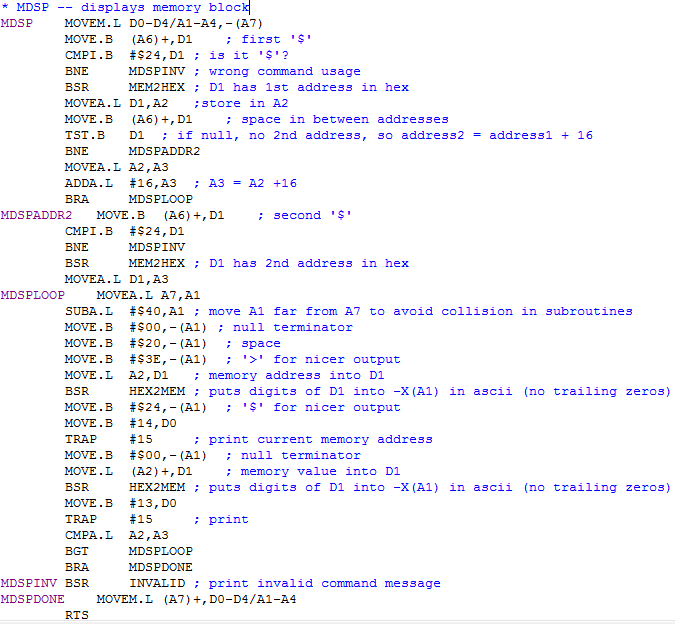
Displays contents of memory between address1 (inclusive) and address2 (exclusive) word by word.

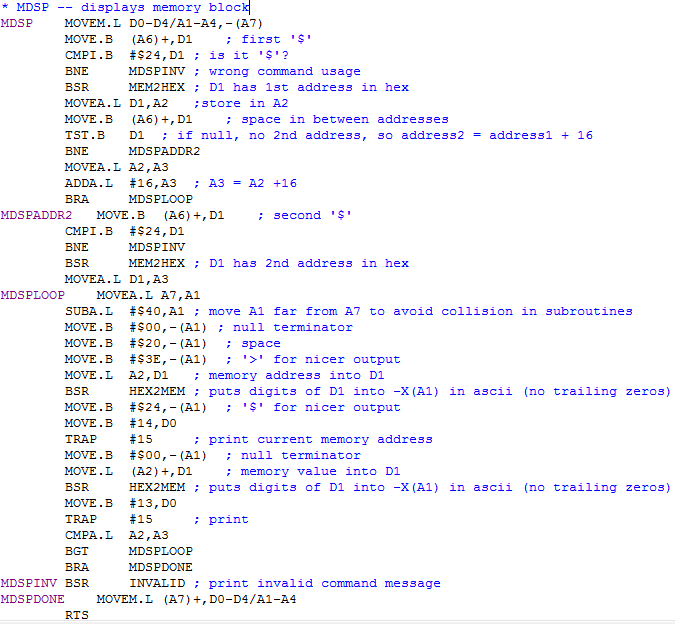
***2.2.2.1-) Debugger Command #2 Algorithm and Flowchart***



*Figure 2.7. Debugger Command #2 Flowchart*

***2.2.2.2-) Debugger Command #2 Assembly Code***





*Figure 2.8. Debugger Command #2 Assembly Code*

***2.2.3-) Debugger Command #3: SORTW***

Sorts a block of memory in between addresses 1 and 2 (inclusive) in either ascending or descending order. The command should be called in the form “SORTW <address1> <address2> A|D”, where A refers to ascending and D to descending (default).

The size of each number within the memory specified is expected to be word, and the type unsigned.

***2.2.3.1-) Debugger Command #3 Algorithm and Flowchart***

The algorithm for sorting is based on Bubble Sort, a method to “bubble up” items to their correct locations. By comparing numbers to the adjacent ones, we can decide whether to swap these or continue. Please refer to Lab Manual 2, Procedure 2.5 for more details.

In addition, a small check was implemented to be able to do either ascending or descending order as requested by the user.

*SORTW // first line*

*Parse input to get ‘start’, ‘end’ and ‘type’ (A or D)*

*While start < end // start will serve as an incrementing pointer*

*If start < start+1 and type = A // using start as an address pointer*

*Swap start with start+1 // so start+1 is the item after start*

*Reset start to original value (start over)*

*Else if start > start+1 and type = D*

*Swap start with start+1*

*Reset start to original value*

*Else // order is fine, move on to next*

*start = start + 1*

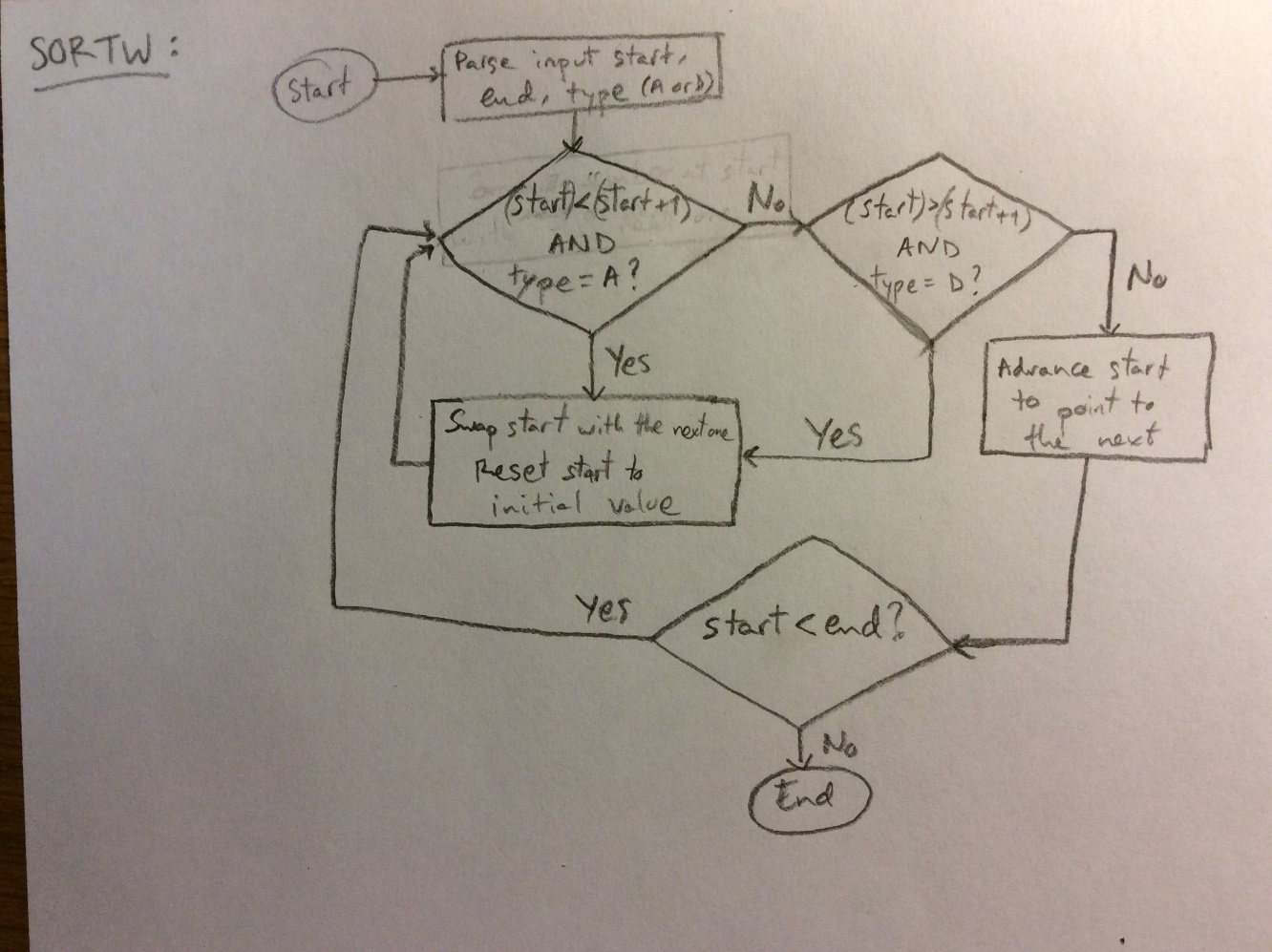
*End if*

*End while*

*Finish // finish*

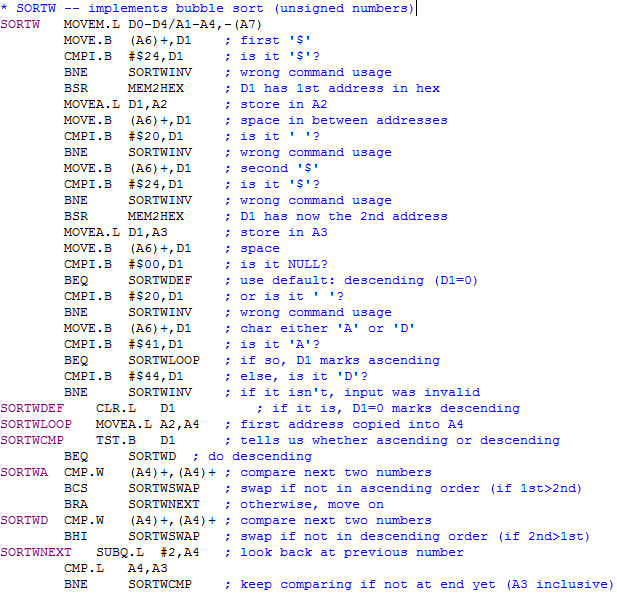
*Figure 2.9. Debugger Command #3 Algorithm*

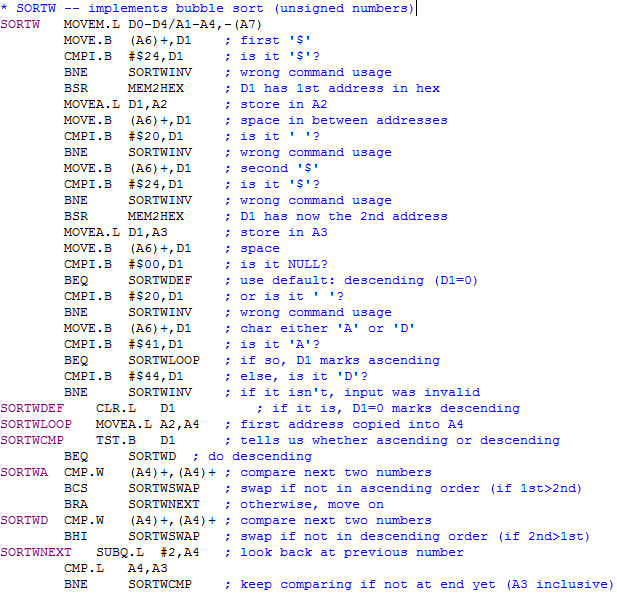
The following flowchart is an abstraction of the algorithm described above:

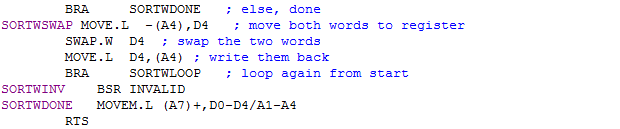


*Figure 2.10. Debugger Command #3 Flowchart*

***2.2.3.2-) Debugger Command #3 Assembly Code***





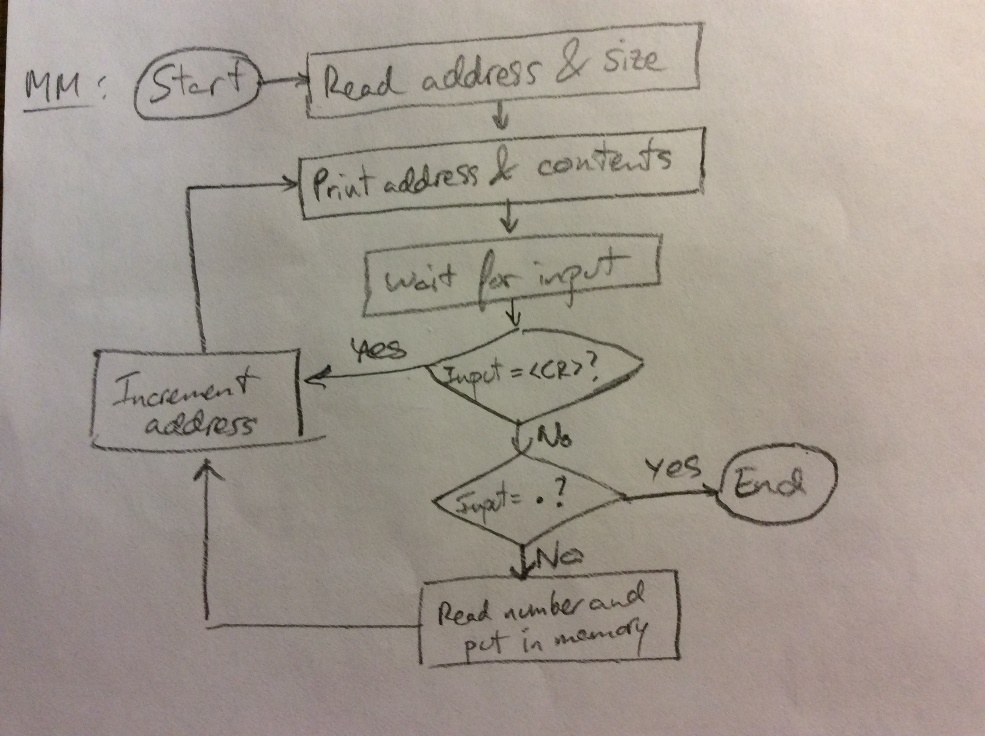


*Figure 2.11. Debugger Command #3 Assembly Code*

***2.2.4-) Debugger Command #4: MM***

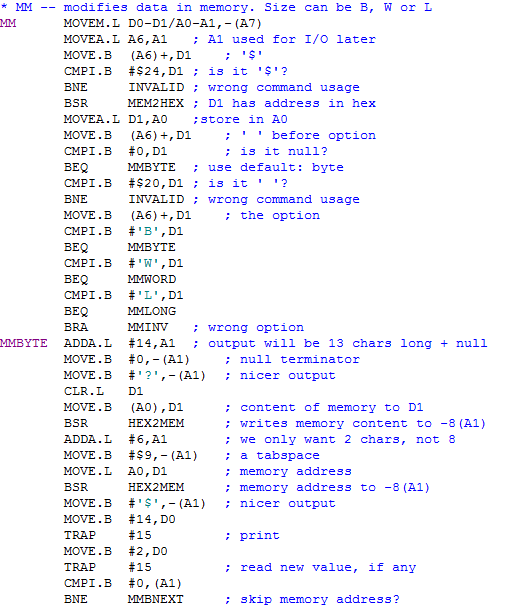
Displays a byte, word or long in memory and allows the user to input a new value in hex. Starts at the address provided and goes on until the user inputs a period ‘.’.

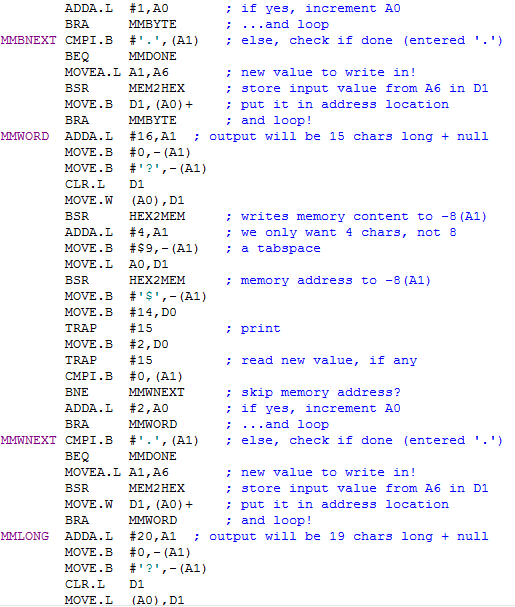
***2.2.4.1-) Debugger Command #4 Algorithm and Flowchart***

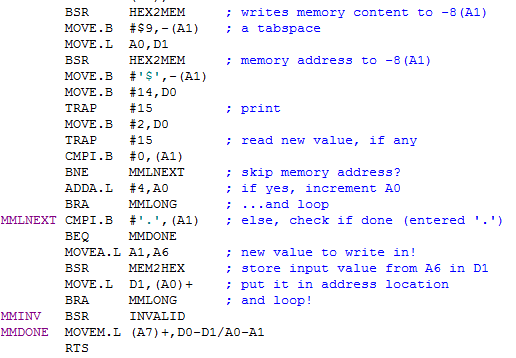
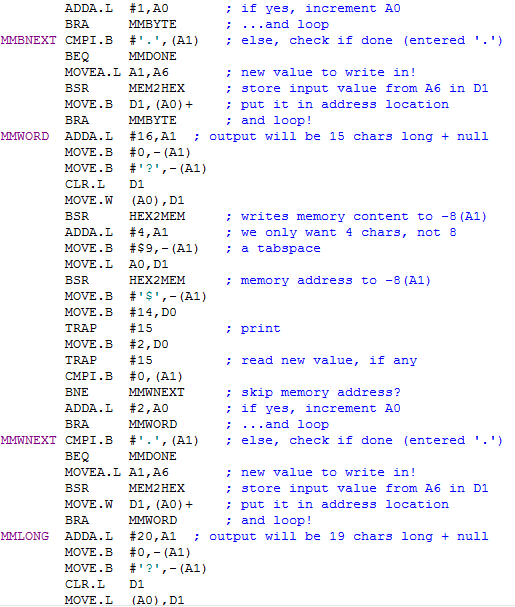


*Figure 2.12. Debugger Command #4 Flowchart*

***2.2.4.2-) Debugger Command #4 Assembly Code***





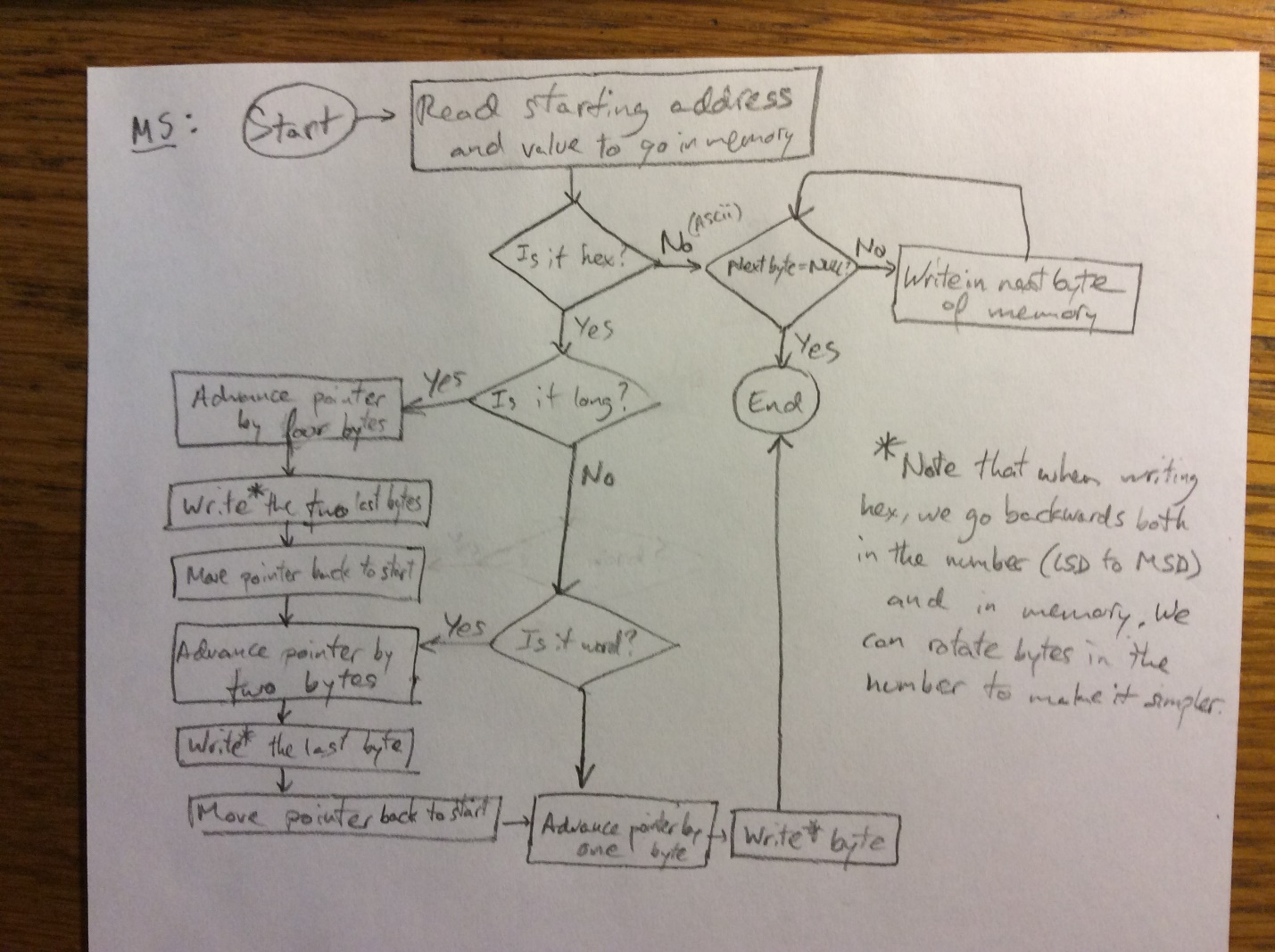


*Figure 2.13. Debugger Command #4 Assembly Code*

***2.2.5-) Debugger Command #5: MS***

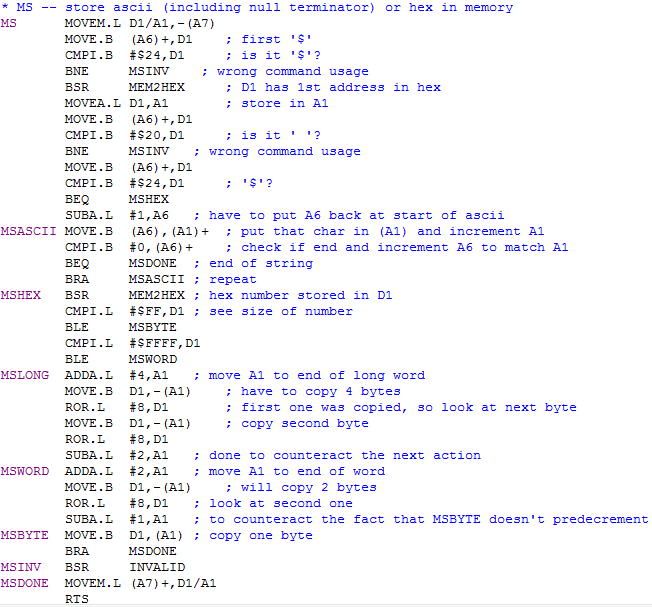
Reads in an ASCII or hex value and places it in memory at the address specified.

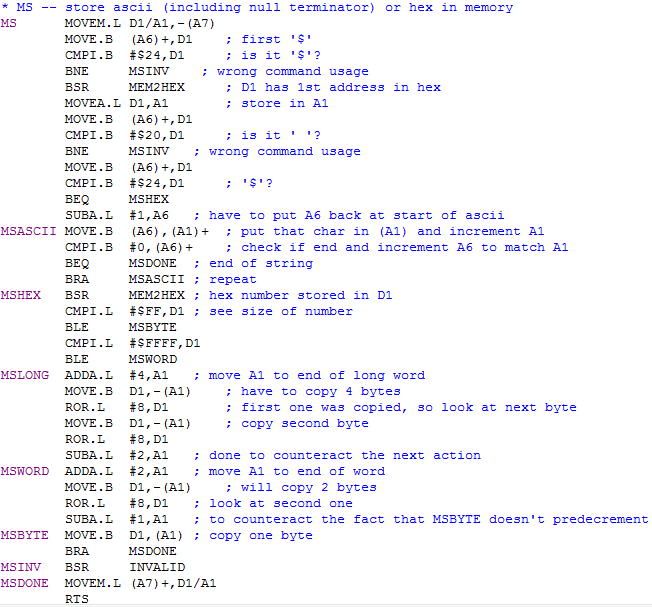
***2.2.5.1-) Debugger Command #5 Algorithm and Flowchart***



*Figure 2.14. Debugger Command #5 Flowchart*

***2.2.5.2-) Debugger Command #5 Assembly Code***



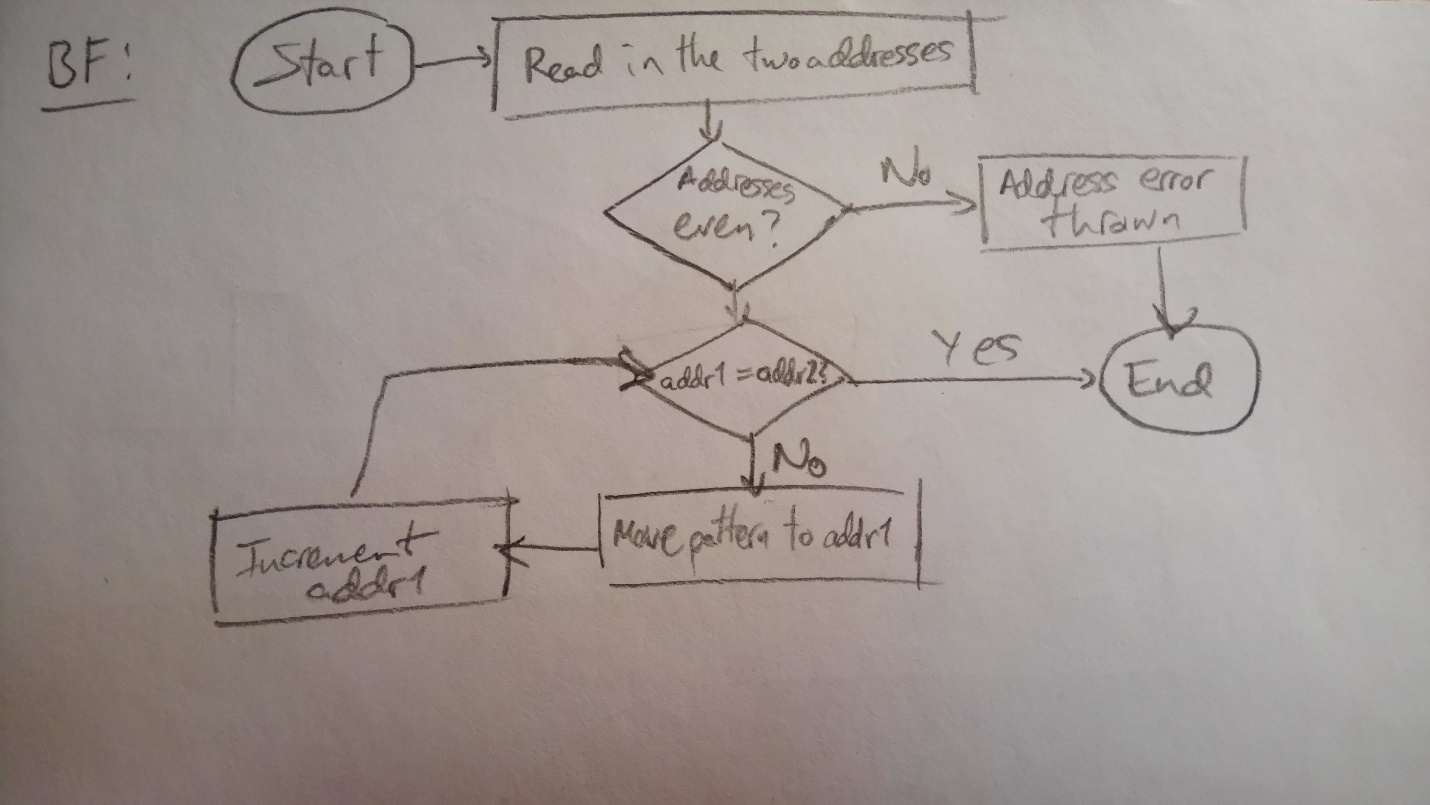


*Figure 2.15. Debugger Command #5 Assembly Code*

***2.2.6-) Debugger Command #6: BF***

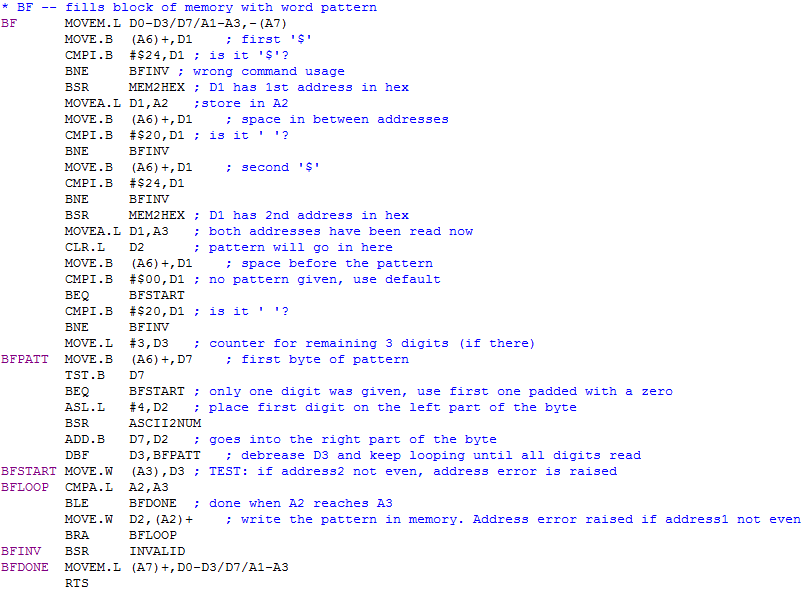
It is similar to 2.2.1

***2.2.6.1-) Debugger Command #6 Algorithm and Flowchart***



*Figure 2.16. Debugger Command #6 Flowchart*

***2.2.6.2-) Debugger Command #6 Assembly Code***



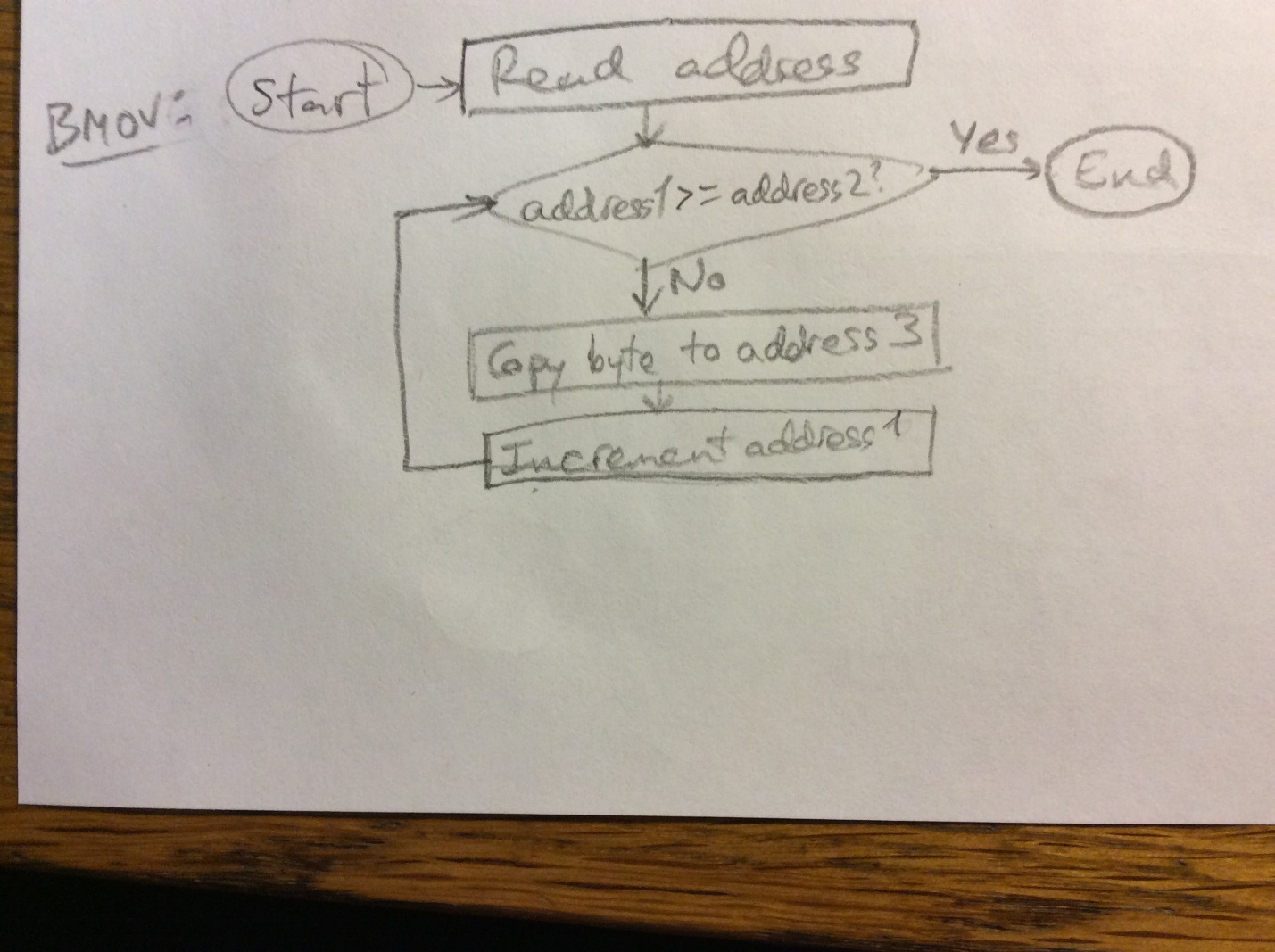
*Figure 2.17. Debugger Command #6 Assembly Code*

***2.2.7-) Debugger Command #7: BMOV***

Moves a block of memory from address1.1 (inclusive) to address1.2 (exclusive) to another place in memory starting at address2.

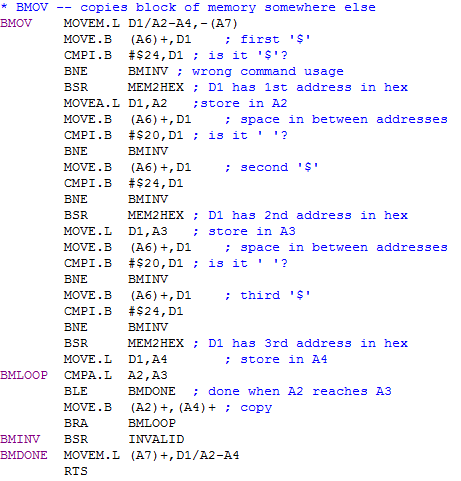
Note that if address1.1 ≤ address2 < address1.2, all data between address2 and address1.2 will be lost because the data between address1.1 and address2 will be repeatedly copied over at that other memory space.

***2.2.7.1-) Debugger Command #7 Algorithm and Flowchart***



*Figure 2.18. Debugger Command #7 Assembly Code*

***2.2.7.2-) Debugger Command #7 Assembly Code***

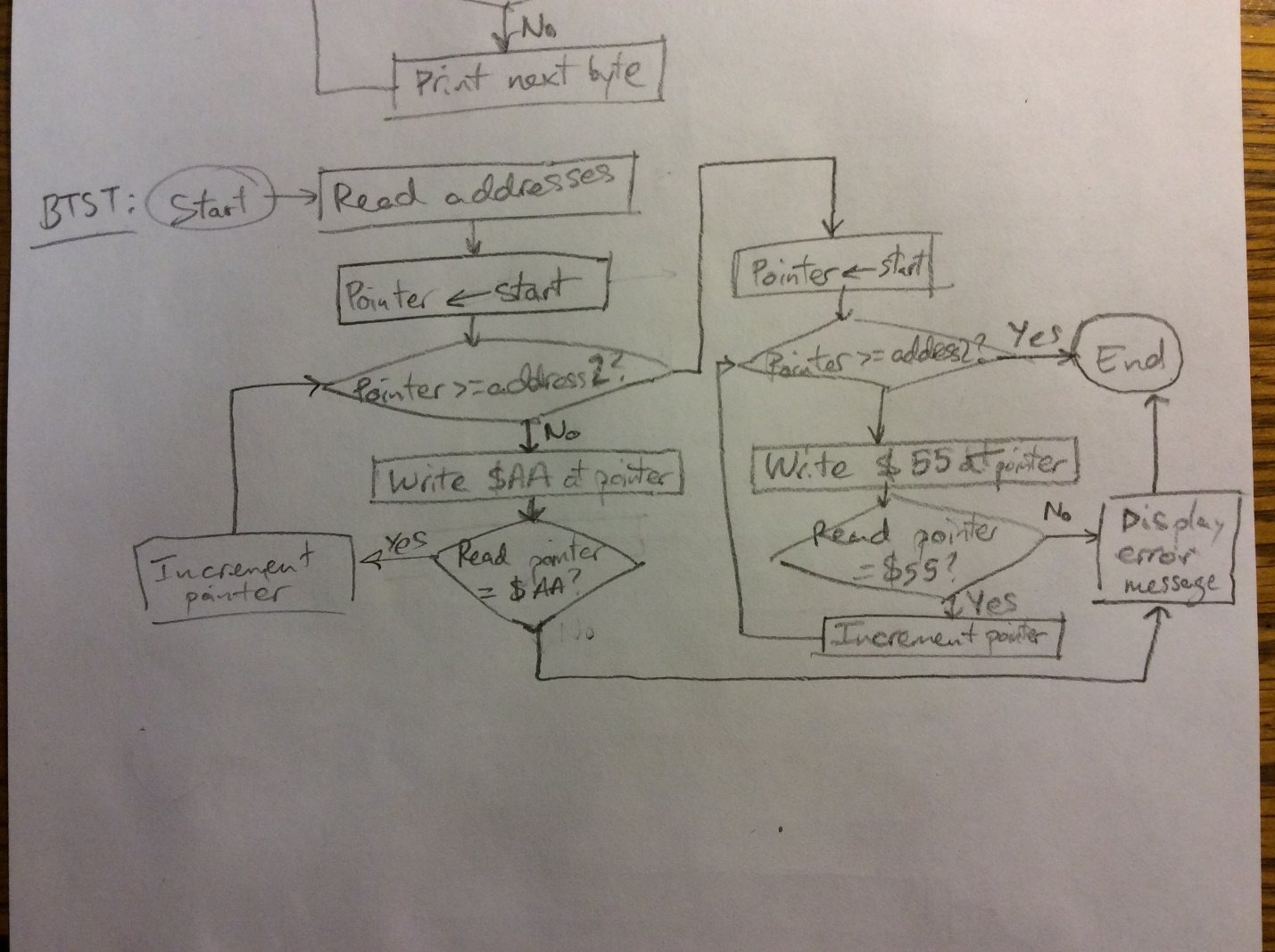


*Figure 2.19. Debugger Command #7 Assembly Code*

***2.2.8-) Debugger Command #8: BTST***

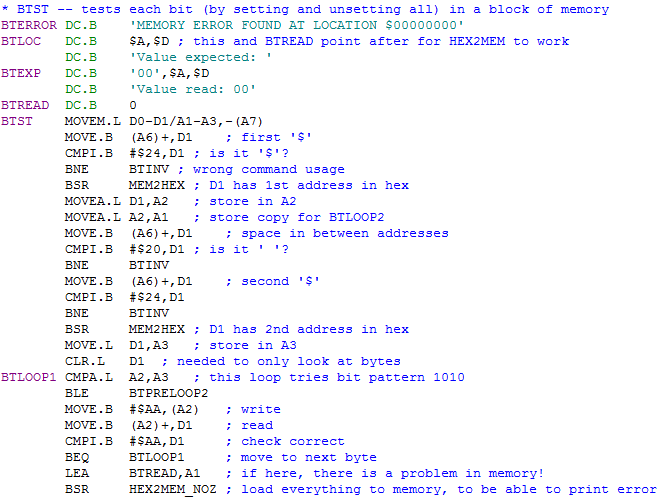
Tests all bits between address1 (inclusive) and address2 (exclusive). This is done by writing and reading the patterns $AA and $55 byte by byte, thus changing each bit. An error is raised and displayed if something else is read after writing.

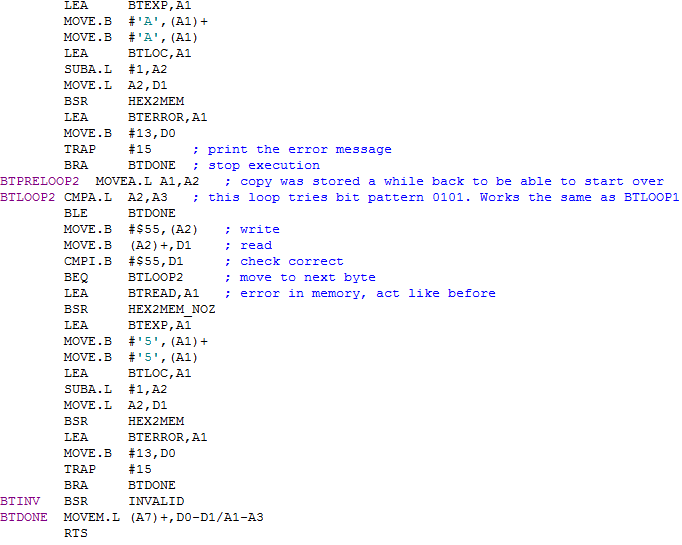
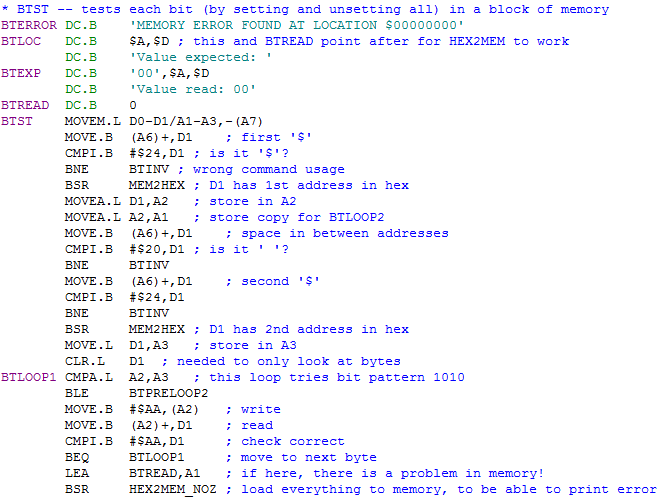
***2.2.8.1-) Debugger Command #8 Algorithm and Flowchart***



*Figure 2.20. Debugger Command #8 Flowchart*

***2.2.8.2-) Debugger Command #8 Assembly Code***



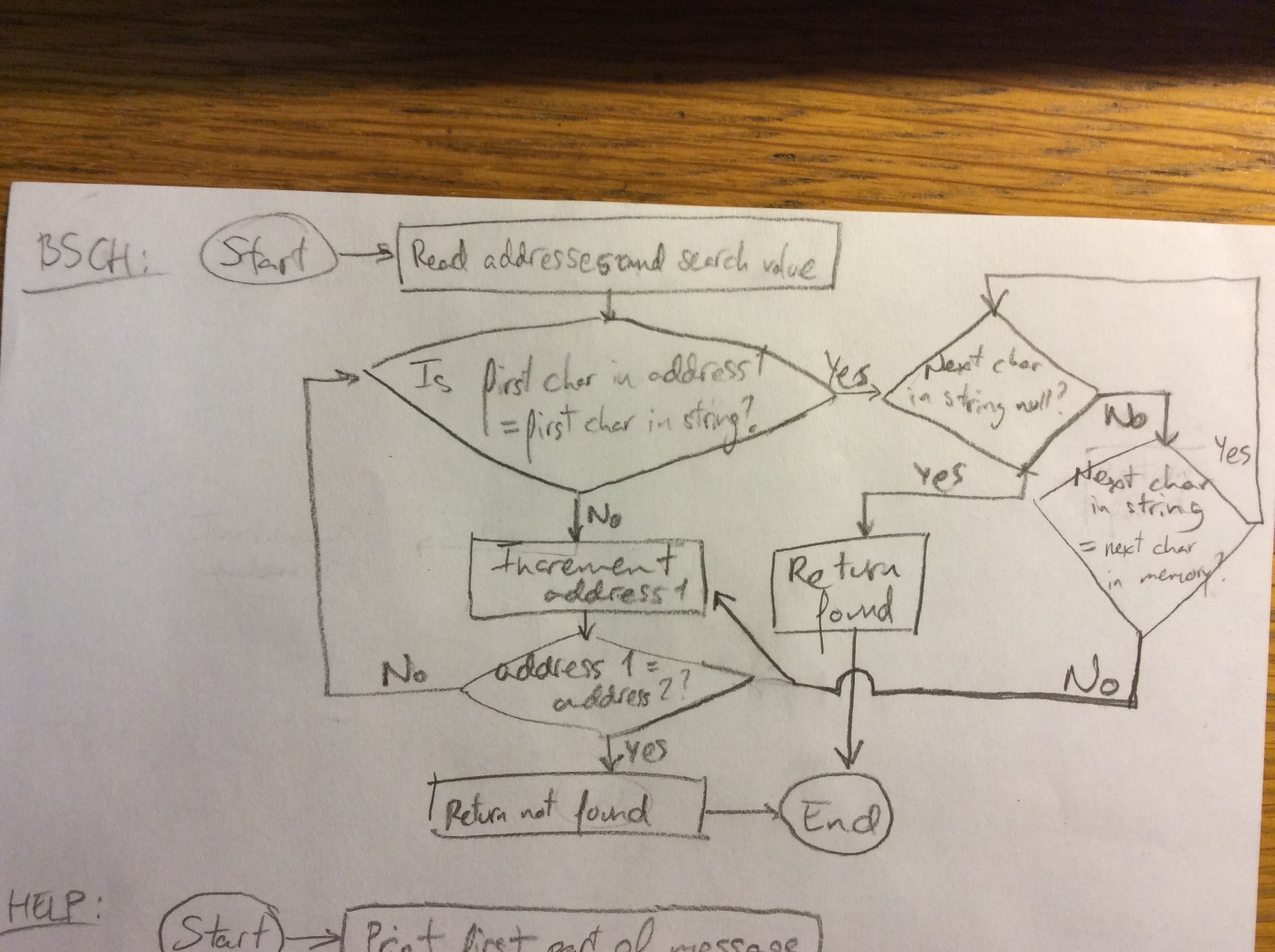


*Figure 2.21. Debugger Command #8 Assembly Code*

***2.2.9-) Debugger Command #9: BSCH***

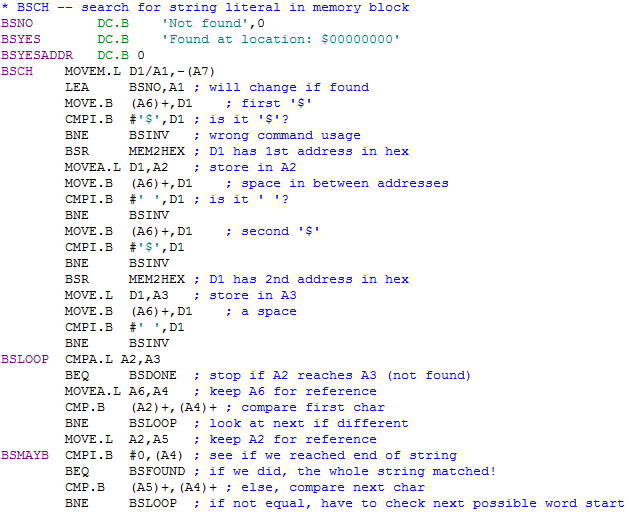
Searches for an ASII string in a block of memory between address1 (inclusive) and address2 (exclusive).

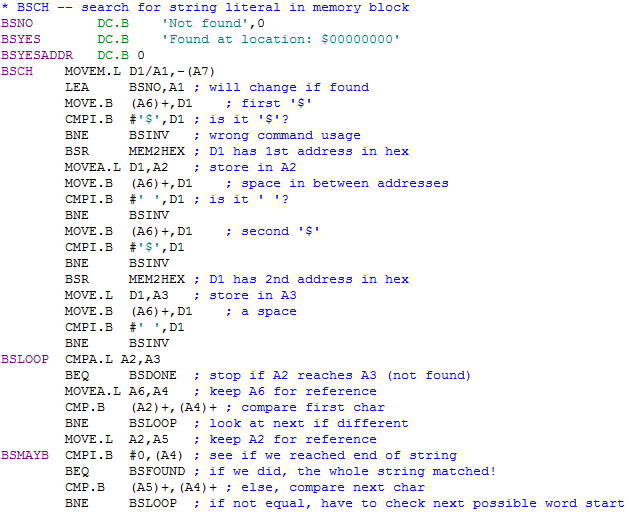
***2.2.9.1-) Debugger Command #9 Algorithm and Flowchart***

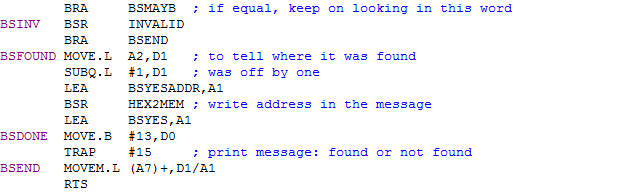


*Figure 2.22. Debugger Command #9 Flowchart*

***2.2.9.2-) Debugger Command #9 Assembly Code***







*Figure 2.23. Debugger Command #9 Assembly Code*

***2.2.10-) Debugger Command #10: GO***

Executes a program stored in some location in memory.

***2.2.10.1-) Debugger Command #10 Algorithm and Flowchart***

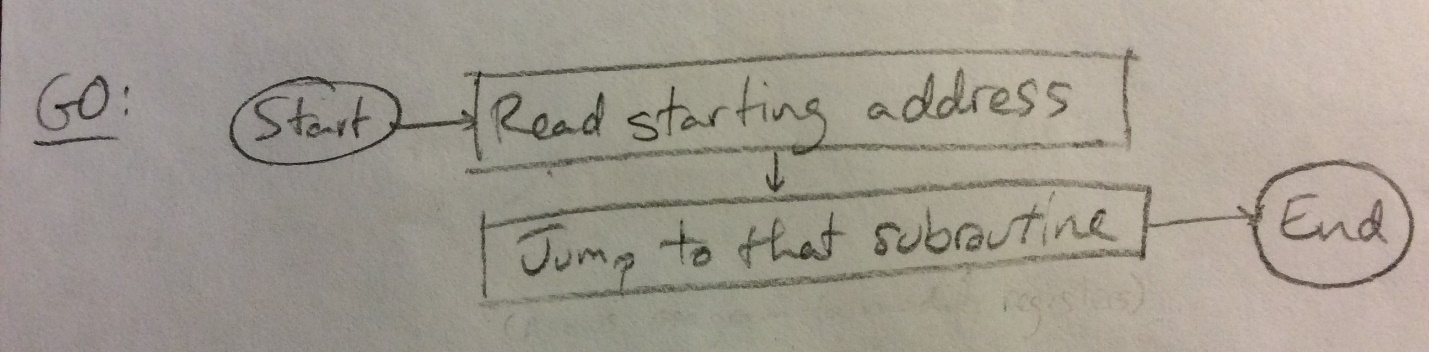
*GO*

*Read starting address from input*

*Jump to that subroutine // execute user’s program*

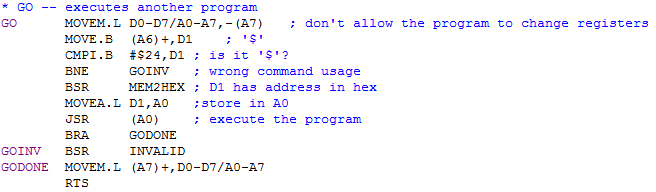
*Finish*

*Figure 2.13. Debugger Command #10 Flowchart*



*Figure 2.24. Debugger Command #10 Flowchart*

***2.2.10.2-) Debugger Command #10 Assembly Code***

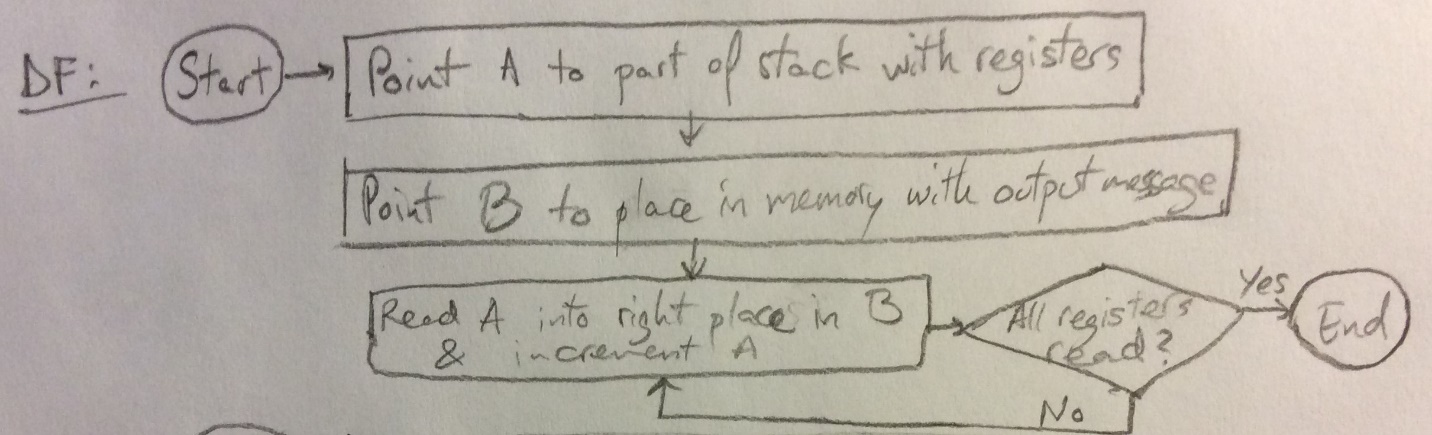


*Figure 2.25. Debugger Command #10 Assembly Code*

***2.2.11-) Debugger Command #11: DF***

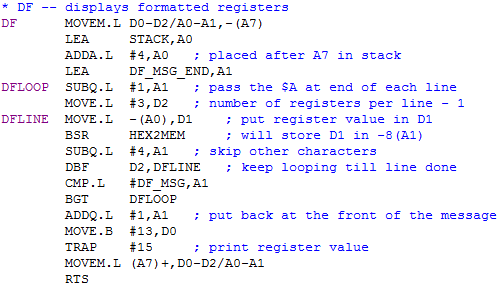
Displays all registers as they were before running the monitor program.

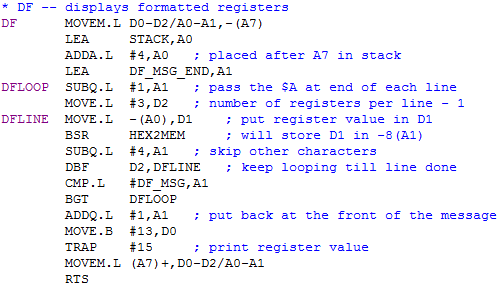
***2.2.11.1-) Debugger Command #11 Algorithm and Flowchart***



*Figure 2.26. Debugger Command #11 Flowchart*

***2.2.11.2-) Debugger Command #11 Assembly Code***



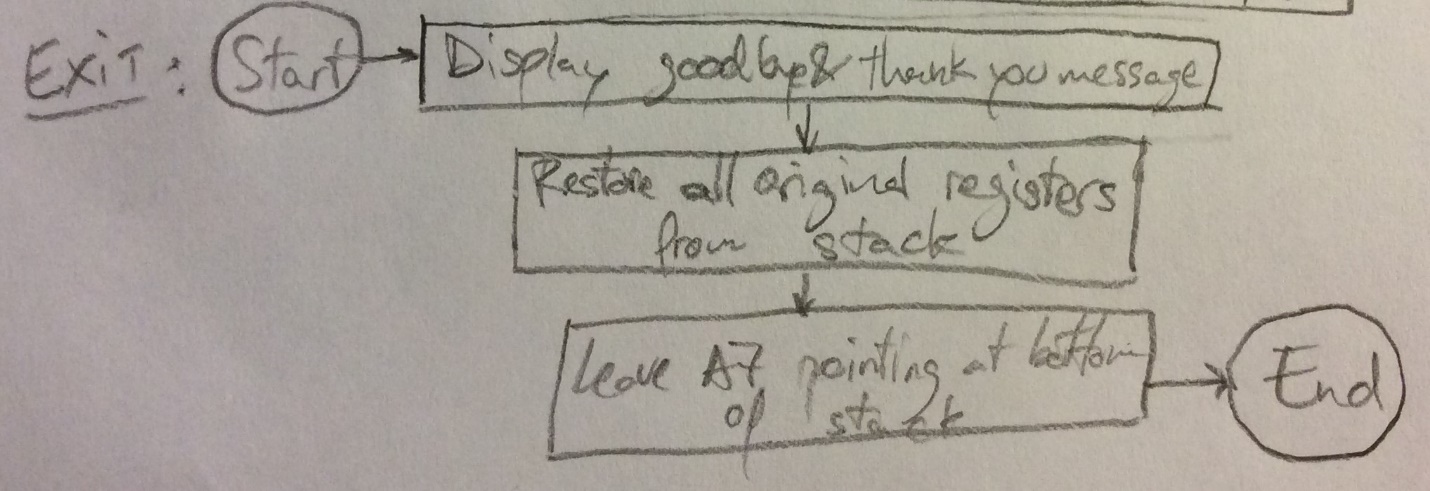


*Figure 2.27. Debugger Command #11 Assembly Code*

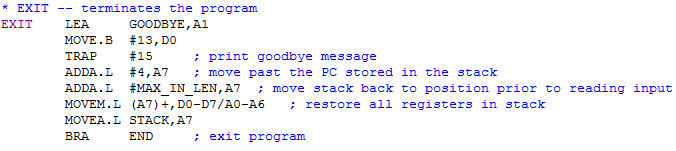
***2.2.12-) Debugger Command #12: EXIT***

Terminates the program and restores the registers to the original values.

***2.2.12.1-) Debugger Command #12 Algorithm and Flowchart***

 *Figure 2.28. Debugger Command #12 Flowchart*

***2.2.12.2-) Debugger Command #12 Assembly Code***

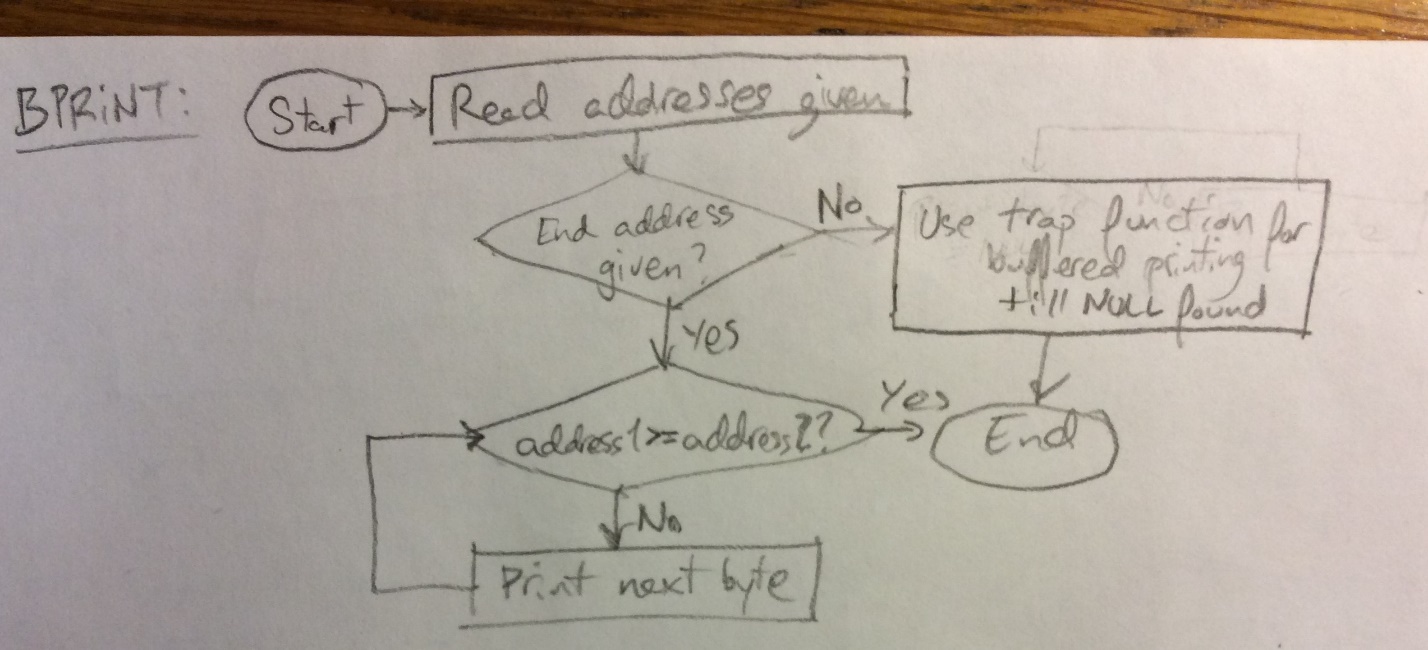


*Figure 2.29. Debugger Command #12 Assembly Code*

***2.2.13-) Debugger Command #13***

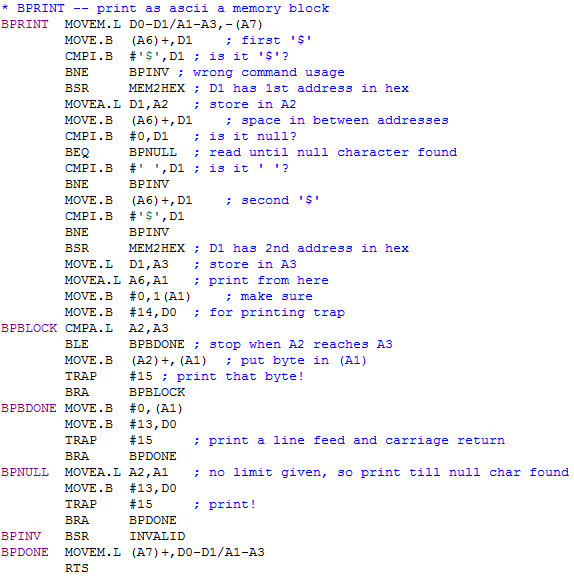
Displays an ascii stored in memory to the console. User may provide an ending address (exclusive) or not. If not provided, the string will be terminated when a null character is found in memory. Starting address is inclusive.

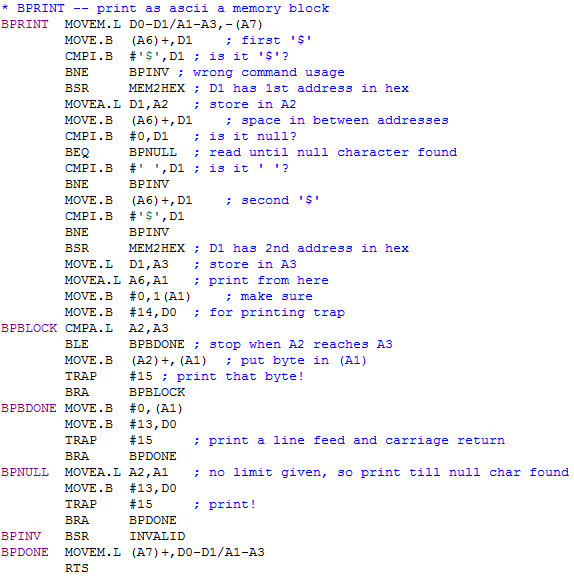
***2.2.13.1-) Debugger Command #13 Algorithm and Flowchart***



*Figure 2.30. Debugger Command #13 Flowchart*

***2.2.13.2-) Debugger Command #13 Assembly Code***



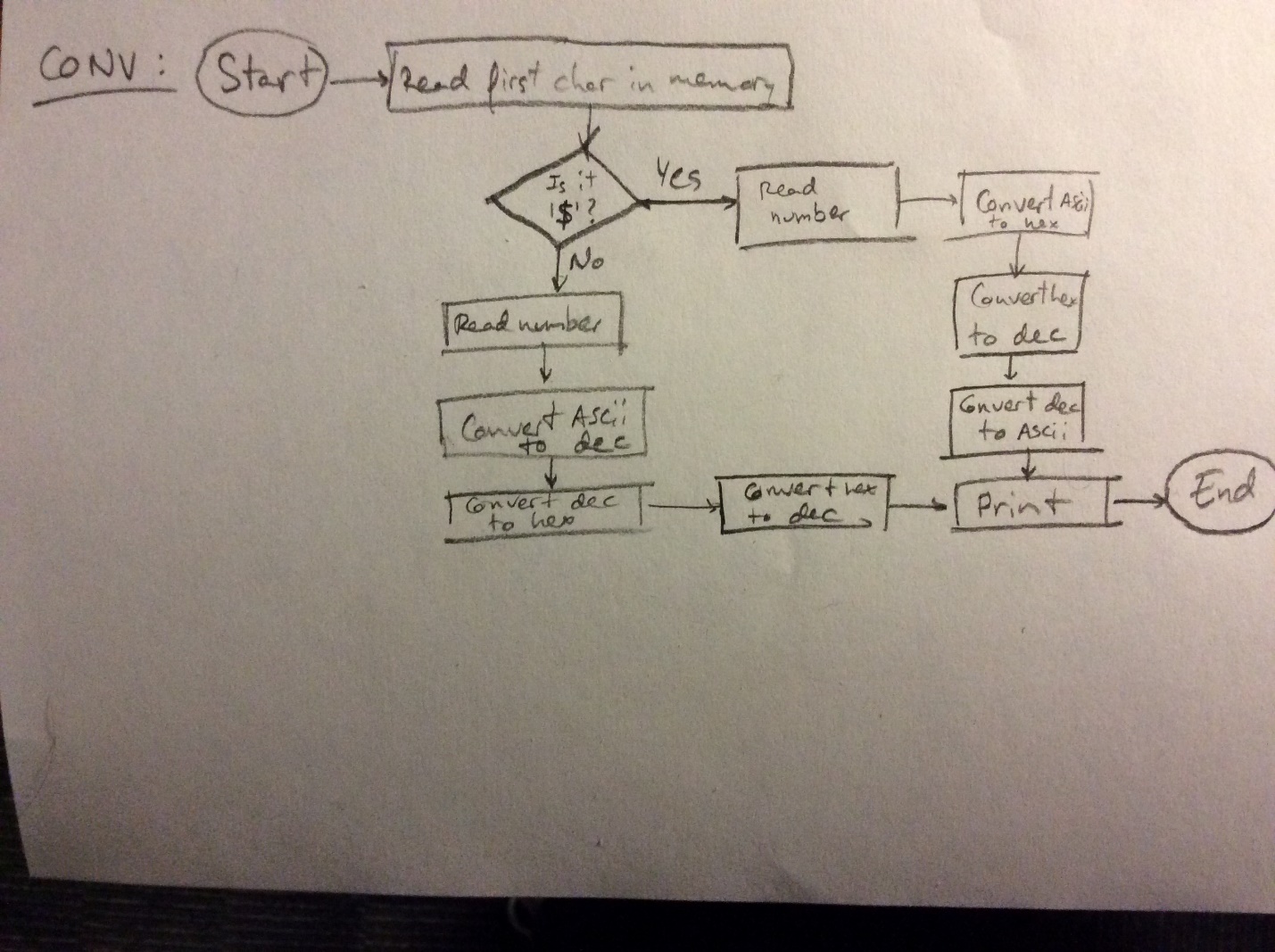


*Figure 2.31. Debugger Command #13 Assembly Code*

***2.2.14-) Debugger Command #14: CONV***

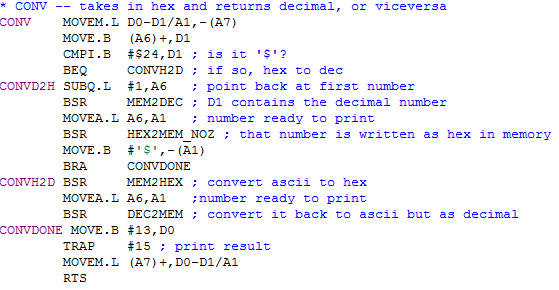
Converts a hex value to (preceded by a dollar sign ‘$’) to decimal, and vice versa.

***2.2.14.1-) Debugger Command #14 Algorithm and Flowchart***



*Figure 2.32. Debugger Command #14 Flowchart*

***2.2.14.2-) Debugger Command #14 Assembly Code***



*Figure 2.33. Debugger Command #14 Assembly Code*

***2.3-) Exception Handlers***

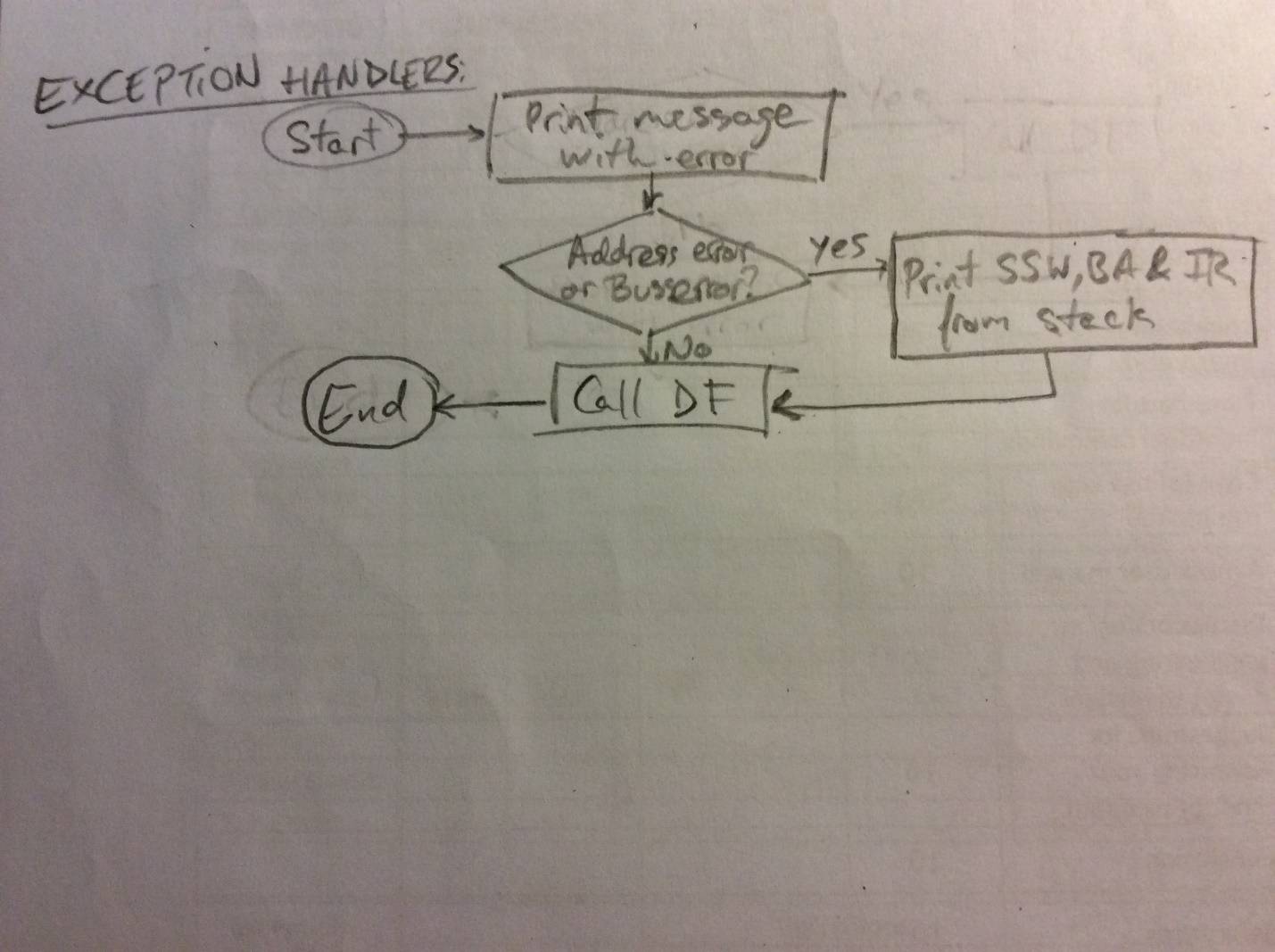
All the exceptions accounted for in this program are:

* Address Error
* Bus Error
* Illegal Instruction
* Privilege Error
* Division By Zero
* Check Error
* Line A Emulator
* Line F Emulator

Since the only difference amongst all exception handlers is the display message identifying the error, they were implemented with common code, except for the message itself. As it is specified in the next sub-sections, all handlers load the appropriate message, and then allow for the common code to also call DF for printing the registers and return to the main routine appropriately.

Nevertheless, the Bus Error and Address Error handlers do differ from the rest in that they also display the SSW, BA and IR. This is added as extra code for these two handlers only.

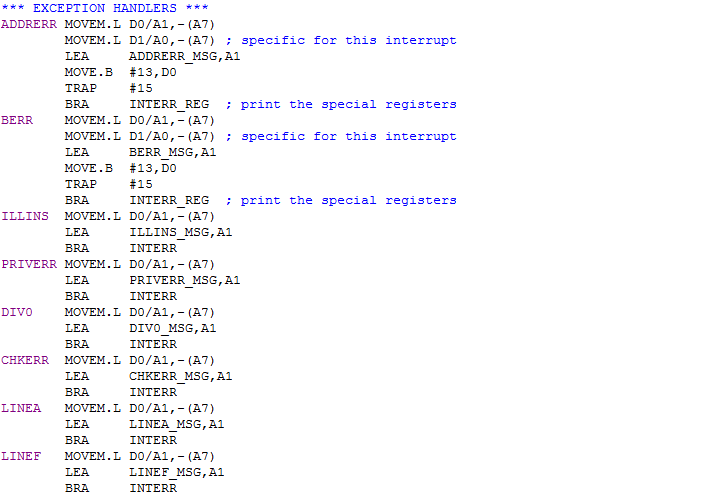
***2.3.1-) Exception Handler Algorithm and Flowchart***

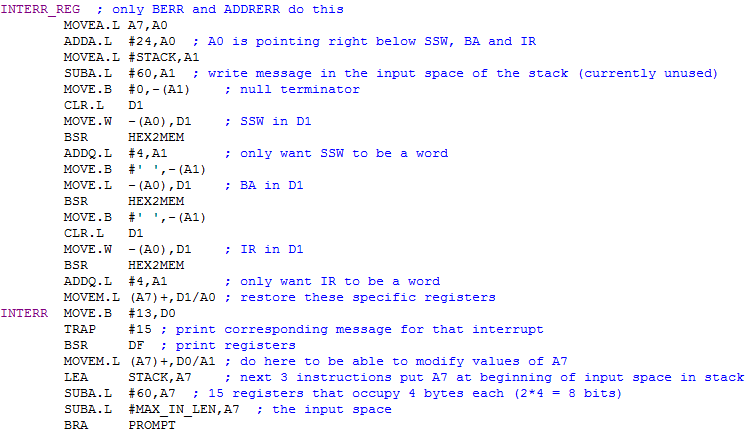


*Figure 2.34. Debugger Command # 1 Flowchart*

***2.3.2-) Exception Handler Assembly Code***

The assembly code for all handlers:





*Figure 2.35. Exception Handling Routines Assembly Code*

***2.4-) Quick User Instruction Manual***

The following text contains a quick user manual with usage for all commands. All addresses must be given in hex, and all hex values must be given with a preceding dollar sign ‘$’. It can be accessed from the running program by executing the ‘HELP’ command:

*HELP: Displays This Message*

*MDSP: Outputs Address And Memory Contents*

*Default address2: address1 + 16*

*MDSP <address1>[ <address2>] eg: MDSP $908 $904<CR>*

*SORTW: Sorts Unsigned Words In A Memory Block*

*Both address1 and address2 are inclusive*

*Default order: descending*

*SORTW <address1> <address2>[ A|D] eg: SORTW $2000 $201E A<CR>*

*MM: Modifies Data In Memory*

*Default: Displays one byte*

*W: Displays one word*

*L: Displays one long word*

*MM <address>[ size]*

*MS: Set Memory To Given ASCII Or Hex*

*Default: ASCII. Prepend $ for hex (byte, word or long)*

*MS <address1> [$]<string|hex> eg: MS $4000 Hello!*

*BF: Fills Block Of Memory With Word Pattern*

*Both addresses must be even*

*Default pattern: 0000*

*If less than 4 digits given, right justified and zero padded*

*BF <address1> <address2>[ pattern] eg: BF $2000 $2200 4325<CR>*

*BMOV: Duplicate A Memory Block At Another Address*

*Must provide two addresses (inclusive, exclusive) for first block*

*Only one address (inclusive start) for second block*

*BMOV <address1.1> <address1.2> <address2>*

*BTST: Test Memory Block*

*BTST <address1> <address2>*

*BSCH: Search In Memory Block*

*BSCH <address1> <address2> <string>*

*GO: Execute Another Program*

*GO <address1>*

*DF: Displays All Formatted Registers eg: DF<CR>*

*EXIT: Exit The Monitor Program eg: EXIT<CR>*

*The two extra commands:*

*BPRINT: Print Block Of Memory*

*Default end: wherever a null char is found*

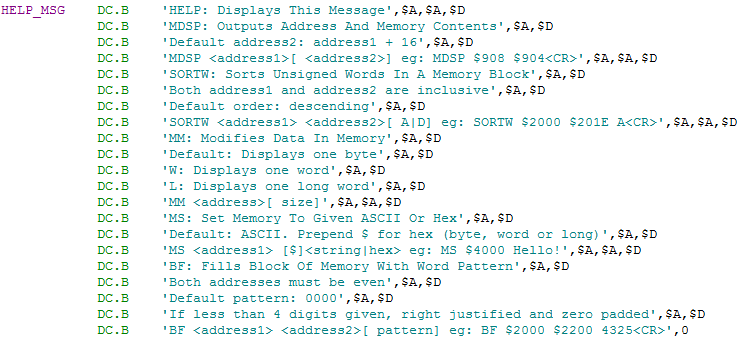
*BPRINT <address1>[ <address2>]*

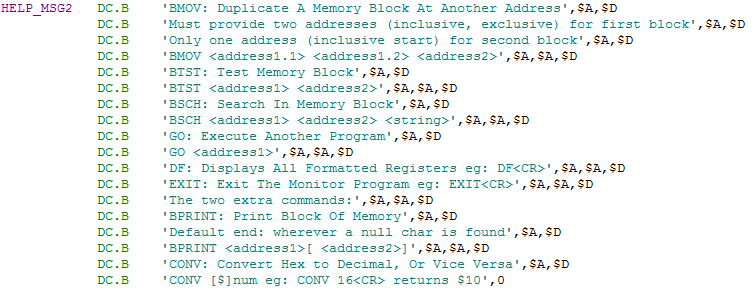
*CONV: Convert Hex to Decimal, Or Vice Versa*

*CONV [$]num eg: CONV 16<CR> returns $10*

***2.4.1-) Assembly Code***

The above message is stored in memory between locations $10FE and $16FB (~1.5KB). The assembly code for it is shown on the next page:





*Figure 2.13. Help Message Assembly Code*

***3-) Discussion***

The design of this monitor program involved a lot of decision taking, much of which can be encountered by a professional engineer almost in a daily basis. Keeping in mind the main goal of producing a fully functional program, many other optimization factors had to be considered throughout both the design and implementation processes. These factors included:

* Speed
* Memory usage
* Simplicity
* Readability of code
* Usability

An example of a decision that took into consideration most of these factors was the implementation of the exception handling routines. In order to keep the code simple and readable and to cut down significantly the use of memory, the common functionality of all these subroutines was identified and implemented as a subroutine on its own. Thereby, each of the handlers could simply perform their specific function, such as loading their particular display message, and then branch onto the common part of the subroutine. This avoided having various similar subroutines for each exception, which would have occupied more memory unnecessarily.

In fact, modularity was a big part to this project. By creating several helper subroutines, such as those for ASCII to hex conversion and vice versa or the one for displaying an invalid command’s message, the code was kept clean and efficient.

Another essential part of the project was the design of the algorithms themselves. Writing pseudocode and flowcharts for each command was definitely good practice in coming up with efficient and well-written code.

In addition, it is worth noting the necessity to consider the user end as well. For this product to be actually useful, it must be usable. Therefore, the format in which the input would be taken was considered carefully and mindfully of the common conventions.

Finally, it must be acknowledged that in this project, as in almost any other production level one, some debugging was required. In this case it was mostly software debugging, but still keeping in mind the hardware, computer architecture and other processor concepts.

***4-) Feature Suggestions***

Firstly, many other commands may be implemented, depending on the users’ needs. Some of these may include loading a program from some other executable file into memory, interchanging memory blocks, etc.

Secondly, the exceptions could be advanced to try to fix the error, or give suggestions on how to fix it, instead of simply acknowledging the error.

Thirdly, the help message could be stored in a more specific region of memory or in external memory, since it occupies a large space and could be an obstacle to other functionality.

Lastly, functionality to interact with peripheral devices could be added.

***5-) Conclusion***

All commands and exception handling routines were implemented successfully. Any user will be capable of executing the previously described functionality. With the help of the Quick User Instruction Manual, it is not hard to get started quickly.

In addition, error handling has been implemented, so that it is difficult and unexpected for the common user to break the code or come across unforeseen hindrances. As long as the commands are used in a logical manner, following the given descriptions, the program will run appropriately.

***6-) References***

[1] T. Harman and D. Hein, “The Motorola MC 68000 Microprocessor Family”, Prentice-Hall Inc., Englewood Cliffs, NJ, 1996.

[2] A. Clements, “Microprocessor Systems Design”, PWS Publishing Company, Boston, MA, 1997.

[3] Educational Computer Board Manual

[4] Experiment 2 Lab Manual

[5] Experiment 3 Lab Manual