

ILLINOIS INSTITUTE OF TECHNOLOGY

ECE 441 Monitor Project

Author:

Adam SUMNER

Teaching Assistant:

Boyang WANG

April 28th, 2015

Acknowledgment

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

Contents

	Page
1 Introduction	5
2 Monitor Program	5
2.1 Command Interpreter	6
2.1.1 Algorithm and Flowchart	6
2.1.2 Assembly Code	9
2.2 Debugger Commands	9
2.2.1 Help	9
2.2.1.1 Algorithm and Flowchart	9
2.2.1.2 Assembly Code	9
2.2.2 Memory Display	9
2.2.2.1 Algorithm and Flowchart	9
2.2.2.2 Assembly Code	10
2.2.3 HXDEC	10
2.2.3.1 Algorithm and Flowchart	10
2.2.3.2 Assembly Code	11
2.2.4 SORTW	11
2.2.4.1 Algorithm and Flowchart	11
2.2.4.2 Assembly Code	12
2.2.5 Memory Modify	12
2.2.5.1 Algorithm and Flowchart	12
2.2.5.2 Assembly Code	13
2.2.6 Memory Set	13
2.2.6.1 Algorithm and Flowchart	13
2.2.6.2 Assembly Code	14
2.2.7 Block Fill	14
2.2.7.1 Algorithm and Flowchart	14
2.2.7.2 Assembly Code	15
2.2.8 Block Move	15
2.2.8.1 Algorithm and Flowchart	15
2.2.8.2 Assembly Code	16
2.2.9 Block Test	16
2.2.9.1 Algorithm and Flowchart	16
2.2.9.2 Assembly Code	17

2.2.10	Block Search	17
2.2.10.1	Algorithm and Flowchart	17
2.2.10.2	Assembly Code	18
2.2.11	Go	18
2.2.11.1	Algorithm and Flowchart	18
2.2.11.2	Assembly Code	19
2.2.12	Display Formatted Registers	19
2.2.12.1	Algorithm and Flowchart	19
2.2.12.2	Assembly Code	20
2.2.13	Modify Register	20
2.2.13.1	Algorithm and Flowchart	20
2.2.13.2	Assembly Code	21
2.2.14	Echo	21
2.2.14.1	Algorithm and Flowchart	21
2.2.14.2	Assembly Code	21
2.3	Exception Handlers	22
2.3.1	Bus Error Exception	22
2.3.1.1	Algorithm and Flowchart	22
2.3.1.2	Assembly Code	23
2.3.2	Address Error Exception	23
2.3.2.1	Algorithm and Flowchart	23
2.3.2.2	Assembly Code	24
2.3.3	Illegal Instruction Error Exception	24
2.3.3.1	Algorithm and Flowchart	24
2.3.3.2	Assembly Code	25
2.3.4	Privilege Violation Error Exception	25
2.3.4.1	Algorithm and Flowchart	25
2.3.4.2	Assembly Code	25
2.3.5	Divide by Zero Error Exception	25
2.3.5.1	Algorithm and Flowchart	25
2.3.5.2	Assembly Code	26
2.3.6	A Line Emulator Error Exception	26
2.3.6.1	Algorithm and Flowchart	26
2.3.6.2	Assembly Code	26
2.3.7	F Line Emulator Error Exception	27
2.3.7.1	Algorithm and Flowchart	27
2.3.7.2	Assembly Code	27
2.3.8	Check Instruction Error Exception	27

2.3.8.1	Algorithm and Flowchart	27
2.3.8.2	Assembly Code	28
2.4	User Instruction Manual Exception Handlers	28
2.4.1	Syntax/Unknown Command Error	28
2.4.1.1	Algorithm and Flowchart	28
2.4.1.2	Assembly Code	29
3	Discussion	29
4	Feature Suggestions	30
5	Conclusion	30

List of Figures

1	Structure of Monitor Program	6
2	Flowchart for Command Line Interpreter	8
3	Flowchart for Help	9
4	Flowchart for Memory Display	10
5	Flowchart for HXDEC	11
6	Flowchart for SORTW	12
7	Flowchart for Memory Modify	13
8	Flowchart for Memory Set	14
9	Flowchart for Block Fill	15
10	Flowchart for Block Move	16
11	Flowchart for Block Test	17
12	Flowchart for Block Search	18
13	Flowchart for Go	19
14	Flowchart for Display Formatted Registers	20
15	Flowchart for Modify Register	21
16	Flowchart for Echo	22
17	Flowchart for Bus Error Exception	23
18	Flowchart for Address Error Exception	24
19	Flowchart for Illegal Instruction Exception	24
20	Flowchart for Privilege Violation Exception	25
21	Flowchart for Divide by Zero Exception	26
22	Flowchart for A Line Emulator Error Exception	26
23	Flowchart for F Line Emulator Error Exception	27
24	Flowchart for Check Instruction Error Exception	28
25	Flowchart for User Instruction Manual Exception Handler	29

Abstract

This project involved designing and implementing a Monitor program using the MC68000 assembly language. The program implements twelve basic debugger functions as well as two author defined functions. It is designed to handle exceptions, and is meant to be an educational piece of software for students taking ECE 441 at the Illinois Institute of Technology.

1 Introduction

The SANPER-1 ELU is a Motorola MC68000 based microcomputer designed by Dr. Jafar Saniie and Mr. Stephen Perich for use in college level computer engineering courses[2]. For user interaction, it utilizes a monitor program called TUTOR that enables users to actively interact with the microcomputer. The design objective of this project is to re-implement the functionality of TUTOR into a student written monitor program titled MONITOR441. The program should be able to perform basic debugger functions such as memory display, memory sort, memory change, etc., and must have the ability to handle exceptions. The design constraints are:

- Code must be smaller than 3K starting from address \$1000
- Stack size must be 1K starting at memory location \$3000
- Macros may not be used
- Erroneous inputs should not kill the program

Twelve debugger functions must be implemented, along with two user defined debugger commands.

2 Monitor Program

The monitor program operates in a command driven environment. It acts as a typical shell, providing a user interface to access the microcomputer's services. The main program being run is a command line interpreter. Based on the input that the user enters, the interpreter determines if the input entered is valid and subsequently executes the specified command. It was

developed using the Easy68K Simulator, thus the TRAP #15 handler is used instead of the MC68000's TRAP #14 handler. The structure of how this program operates is shown in Figure 1.

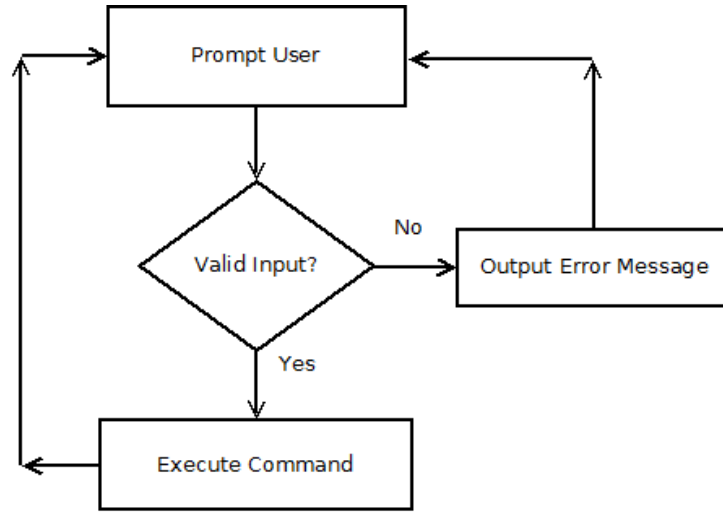


Figure 1: Structure of Monitor Program

2.1 Command Interpreter

2.1.1 Algorithm and Flowchart

The algorithm for the command interpreter uses simple string matching to determine if input is correct. The algorithm begins by outputting the message `MONITOR441>` and accepting input from the user. It then checks for the ASCII value \$48 which corresponds to the letter H. This is to check for either the `HELP` command or `HXDC` command. If an H was not entered, it then checks for the ASCII value \$4D which corresponds to a memory command. If this fails, then it checks for ASCII value \$47, corresponding to the `GO` command. If this fails, the ASCII value \$44 is tested, corresponding to the `DF` command. If this fails, it checks for \$42, which signifies a `BLCK` command. If this fails, \$53 is tested for the `SORTW` command. If this fails, \$45 is tested for the `ECHO` command. If this fails \$2E is checked for the modify register command. If all of these checks fail, the user has entered incorrect input and an error message is displayed. If any of these checks succeed, the command line interpreter jumps to the respective command's helper interpreter function.

These subroutines check for each character of the user input in order to verify the command the user entered was correct. These helper functions also serve to differentiate commands that start with the same character. The flowchart for this process is shown in Figure 2.

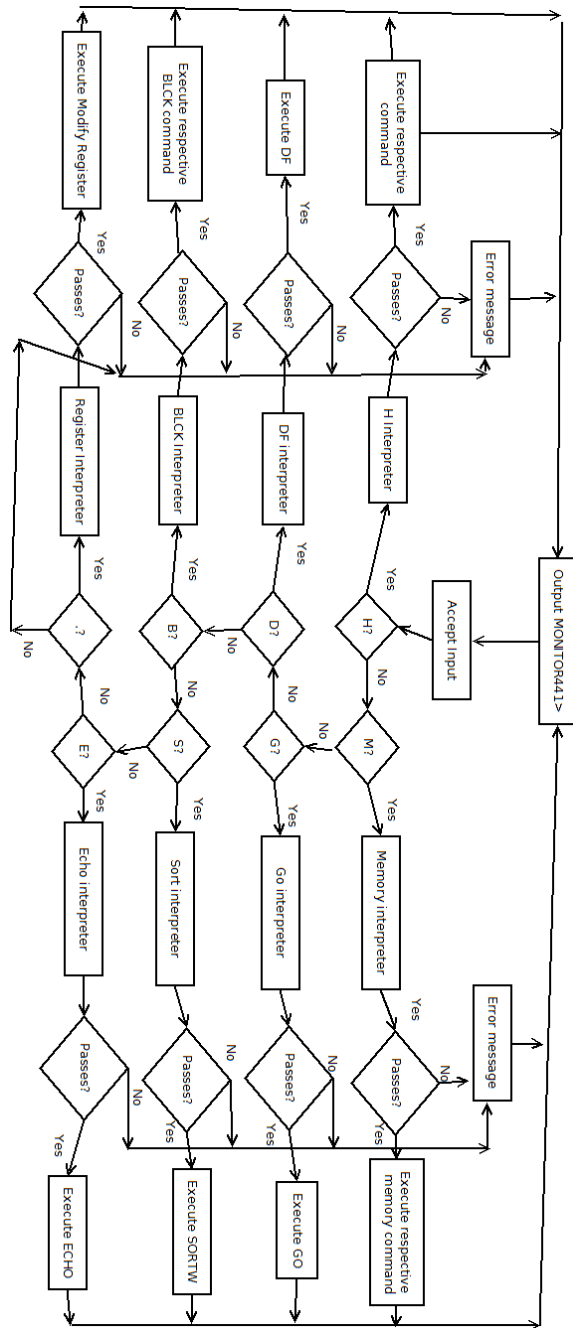


Figure 2: Flowchart for Command Line Interpreter

2.1.2 Assembly Code

2.2 Debugger Commands

2.2.1 Help

2.2.1.1 Algorithm and Flowchart

Help is a simple command that prints out a series of strings that display the available commands, their syntax, and a short description of each command. The syntax to invoke this command is `HELP`. The flowchart for this command is shown in Figure 3.

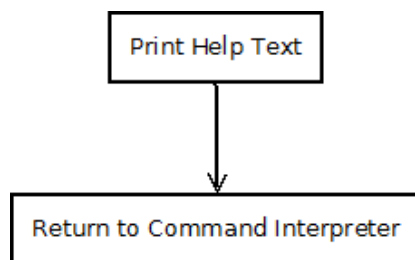


Figure 3: Flowchart for Help

2.2.1.2 Assembly Code

2.2.2 Memory Display

2.2.2.1 Algorithm and Flowchart

Memory display is an extremely useful tool to look at blocks of memory. The syntax to call this function is `MDSP <address1> <address2>`, where `<address1>` is the starting address and `<address2>` is the ending address of the memory contents to be shown. This command also displays the block of memory from `<address1>` to `<address2 +16bytes>`. The flowchart for this command is shown in Figure 4.

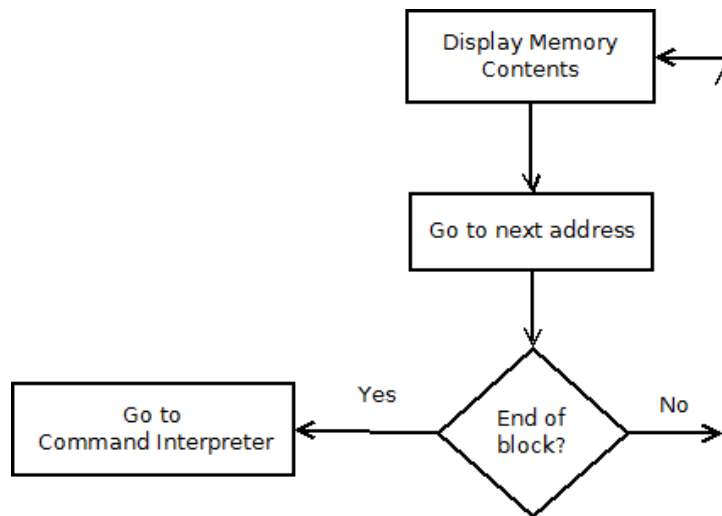


Figure 4: Flowchart for Memory Display

2.2.2.2 Assembly Code

2.2.3 HXDEC

2.2.3.1 Algorithm and Flowchart

This command allows the user to enter a hexadecimal value (up to FFFF), and the program will return the equivalent value in decimal format. The syntax to call this function is `HXDEC <data>`. It works by extracting the ASCII values byte by byte and determining the 16's place of each byte. The value extracted is then multiplied by its respective 16's place and added to a register that stores the total. This total must then be converted into BCD for output and then into ASCII to display it on the terminal. The flowchart for this command is shown in Figure 5.

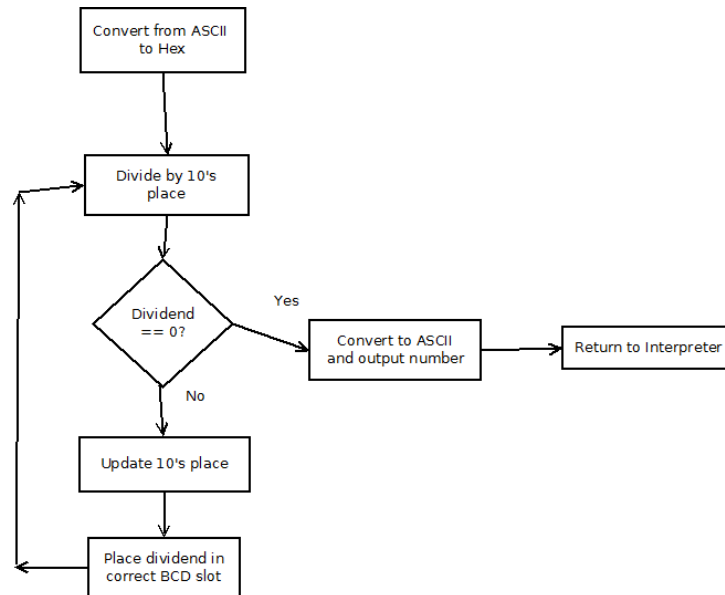


Figure 5: Flowchart for HXDEC

2.2.3.2 Assembly Code

2.2.4 SORTW

2.2.4.1 Algorithm and Flowchart

This command implements the most common sort algorithm for a set of data, the bubble sort. Because the user has the choice to choose between sorting the data in ascending or descending order, it also implements a “rock” sort. It works by first determining which option, ascending or descending, the user has selected. Once determined, the first data in the set is analyzed to the next immediate adjacent value in memory. If the current data is larger than the next data (assuming ascending order for example), the two words of data are swapped. This value is continuously checked against its immediate adjacent memory until it “fits” in the current state of the list. This process is repeated for n elements in a list of n words. The runtime is $\mathcal{O}(n^2)$, and the syntax for this command is `SORTW <option> <address1> <address2>`, where both `<address1>` and `<address2>` are even addresses. The flowchart is shown in Figure 6.

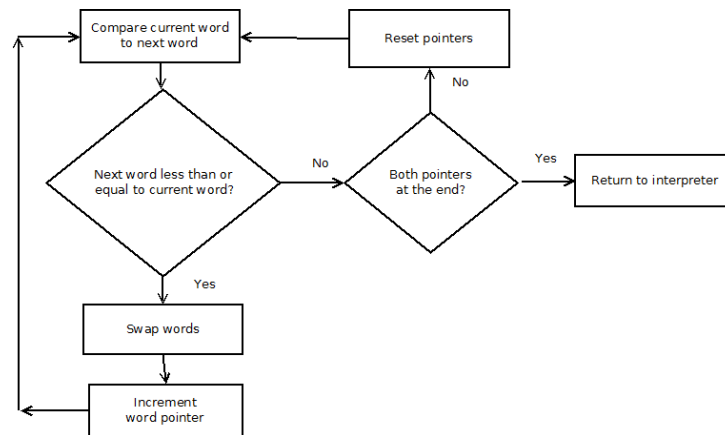


Figure 6: Flowchart for SORTW

2.2.4.2 Assembly Code

2.2.5 Memory Modify

2.2.5.1 Algorithm and Flowchart

This command first determines which option the user has selected. Depending on this option, it reads the address entered by the user and displays the specified amount of data currently stored in memory. The user is then prompted to enter data to store into memory. The command increments the memory location and asks for input until the user enters the '.' character. The syntax for this command is `MM <option> <address>`. The flowchart is shown in Figure 7.

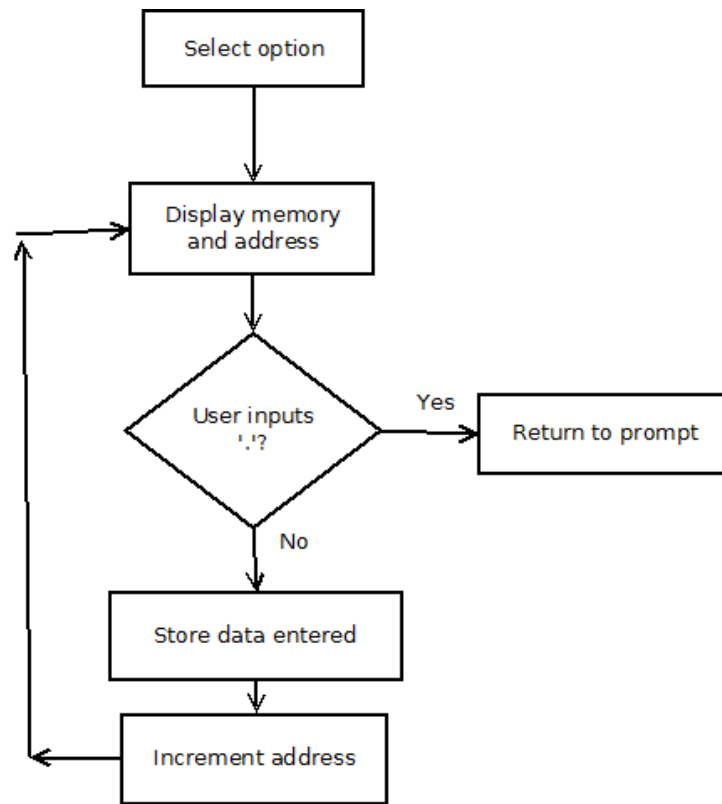


Figure 7: Flowchart for Memory Modify

2.2.5.2 Assembly Code

2.2.6 Memory Set

2.2.6.1 Algorithm and Flowchart

This command is a simpler version of Memory Modify. It parses the data the user entered and stores it at one specified address. It has the syntax **MS** **<data>** **<address>**. The data entered must be byte sized. The flowchart is shown in Figure 8.

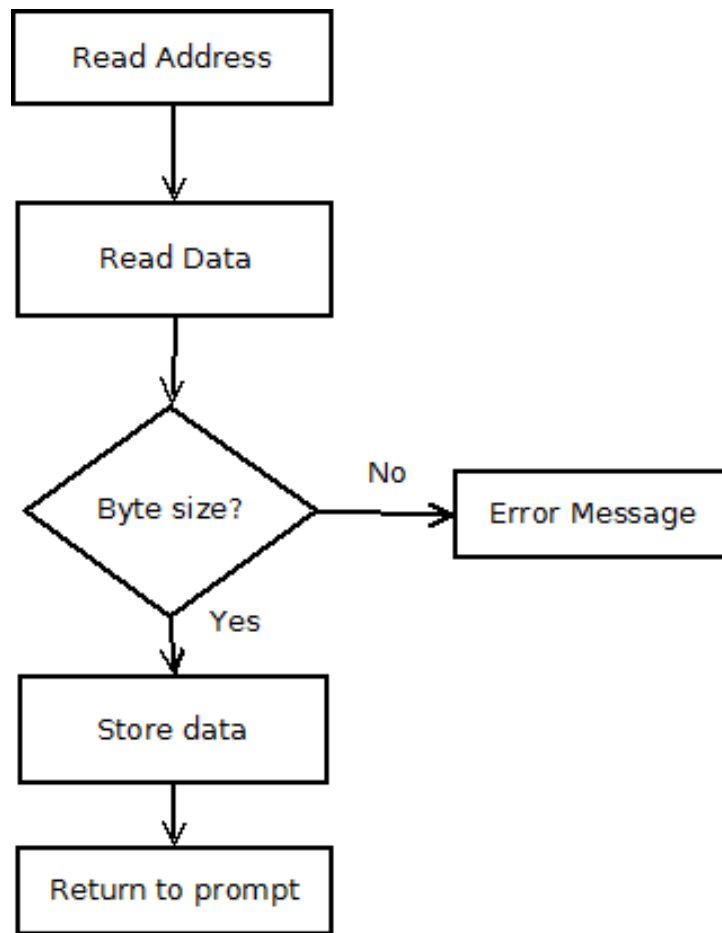


Figure 8: Flowchart for Memory Set

2.2.6.2 Assembly Code

2.2.7 Block Fill

2.2.7.1 Algorithm and Flowchart

This command requires two even addresses to be entered. It then parses the word sized data entered by the user and fills the block of memory from the first address to the second address. The syntax for this command is **BF** `<data> <address1> <address2>`. The flowchart is shown in Figure 9.

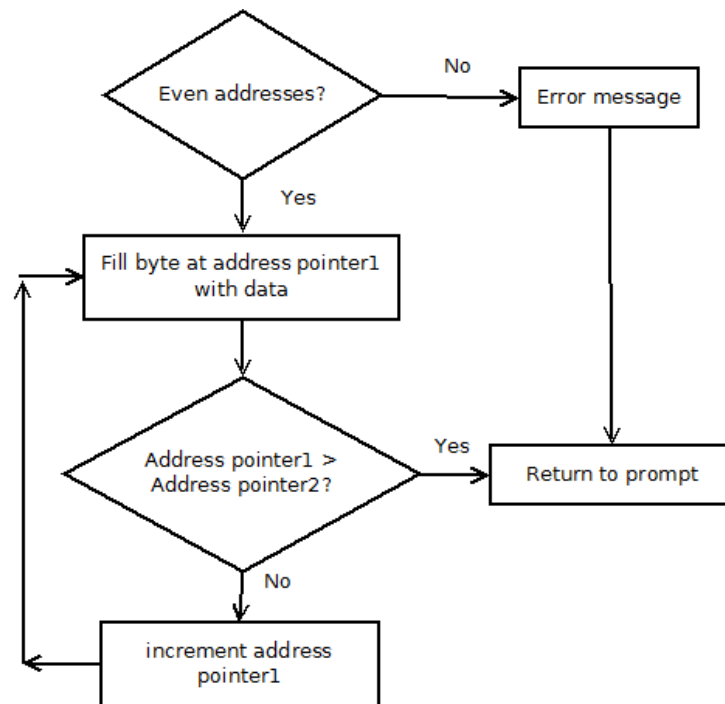


Figure 9: Flowchart for Block Fill

2.2.7.2 Assembly Code

2.2.8 Block Move

2.2.8.1 Algorithm and Flowchart

This command move a block of memory from one section to another. Both block sizes must be equal. Starting from the first address of the first block and the first address of the second block, it moves data byte by byte to the respective memory locations until all data has been copied. Its syntax is `BMOV <address1> <address2> <address3> <address4>`. The flowchart is shown in Figure 10.

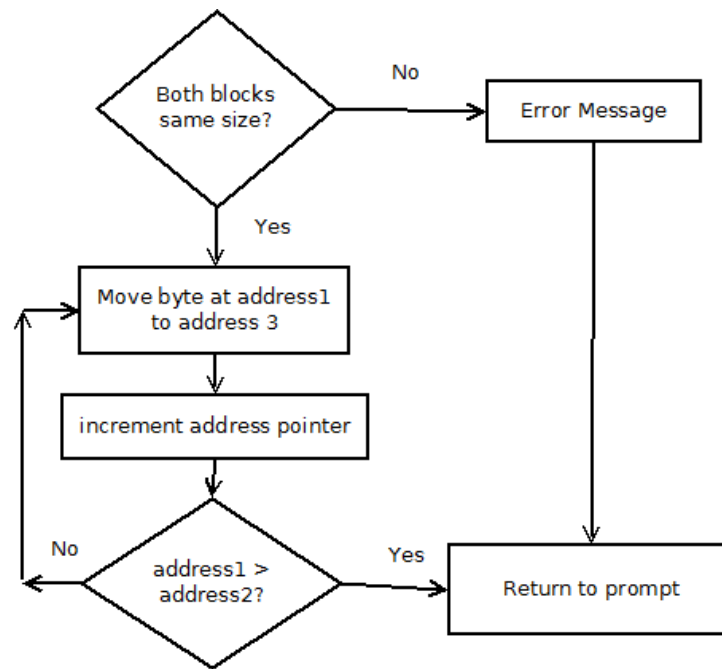


Figure 10: Flowchart for Block Move

2.2.8.2 Assembly Code

2.2.9 Block Test

2.2.9.1 Algorithm and Flowchart

This command fills a block of memory with byte sized data, then checks each byte of the block. If any byte is not equal to the data originally written, the program outputs the data read and the address where the test failed. If no error is detected, the program outputs a message declaring the test passed. The syntax for this command is `BTST <data> <address1> <address2>`. The flowchart is shown in Figure 11.

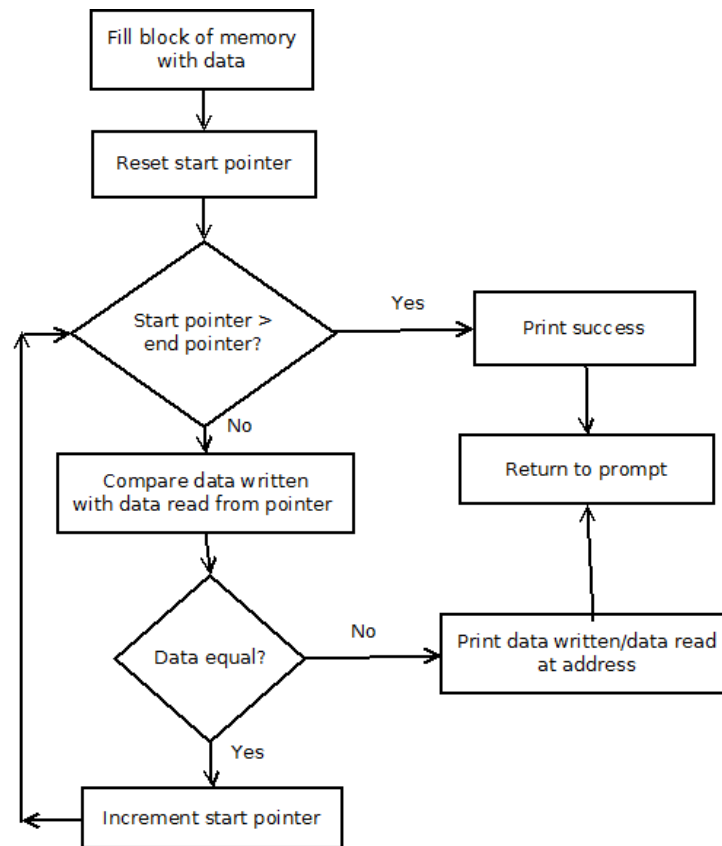


Figure 11: Flowchart for Block Test

2.2.9.2 Assembly Code

2.2.10 Block Search

2.2.10.1 Algorithm and Flowchart

This command searches through a block of memory for data entered by the user. It does so by checking each value in memory byte by byte. The syntax for this command is BSCH <data> <address1> <address2>. The flowchart is shown in Figure 12.

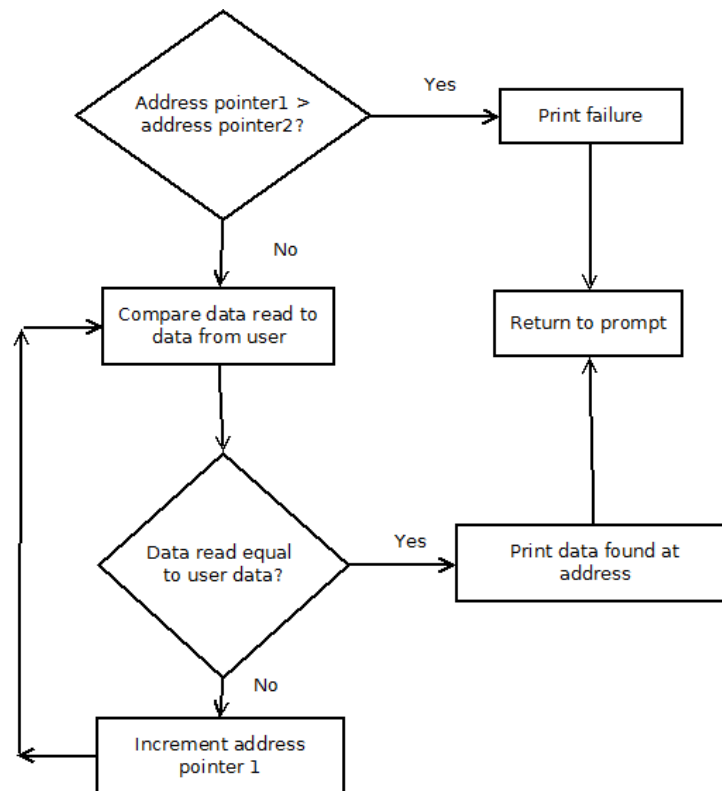


Figure 12: Flowchart for Block Search

2.2.10.2 Assembly Code

2.2.11 Go

2.2.11.1 Algorithm and Flowchart

This command jumps to an address in memory and executes the machine code stored at that address. The syntax is `GO <address>`. The flowchart is shown in Figure 13.

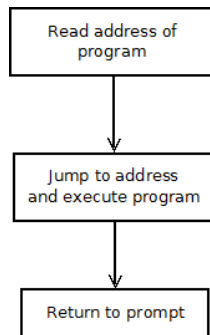


Figure 13: Flowchart for Go

2.2.11.2 Assembly Code

2.2.12 Display Formatted Registers

2.2.12.1 Algorithm and Flowchart

This command displays the values of the registers as well as the stack pointers and program counter. It does so by first popping these values which were previously stored on stack item by item. They are then converted to ASCII for output and displayed on the terminal. The syntax is **DF**. The flowchart is shown in Figure 14.

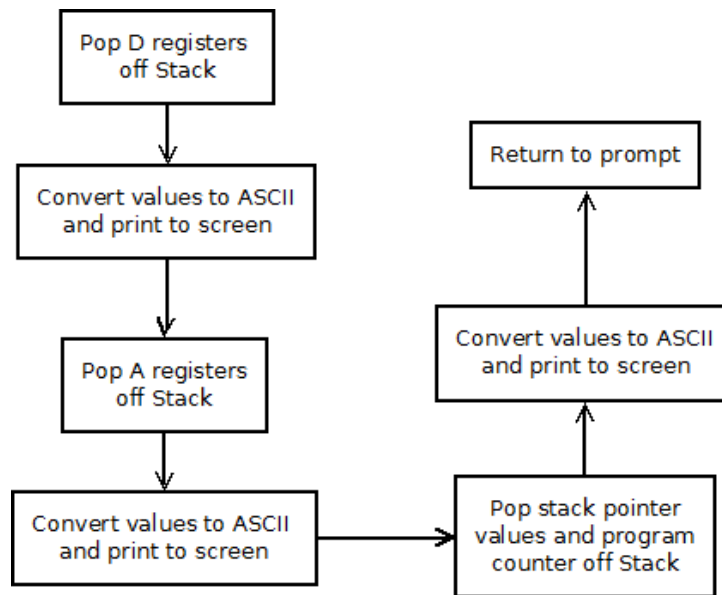


Figure 14: Flowchart for Display Formatted Registers

2.2.12.2 Assembly Code

2.2.13 Modify Register

2.2.13.1 Algorithm and Flowchart

This command is used to change the value of a specific A or D register. This is done by parsing the data entered by the user, then updating the current value of the selected register. The syntax is `.<Register Type> <data>`. The flowchart is shown in Figure 15.

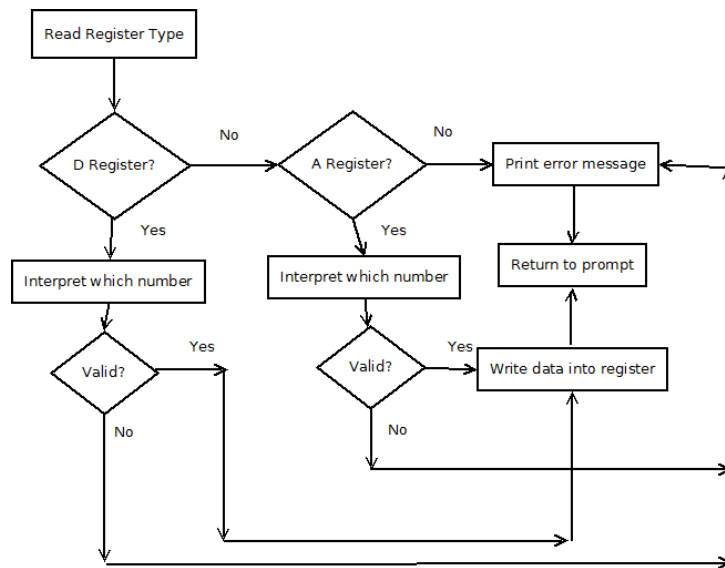


Figure 15: Flowchart for Modify Register

2.2.13.2 Assembly Code

2.2.14 Echo

2.2.14.1 Algorithm and Flowchart

This is a simple command that outputs what the user inputs. This is done by parsing the data entered by the user and immediately setting up a trap I/O call that outputs what was just entered. The syntax is `ECHO <data>`. The flowchart is shown in Figure 16.

2.2.14.2 Assembly Code

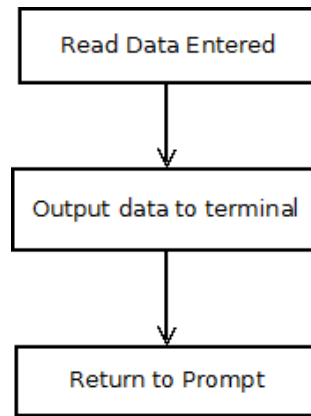


Figure 16: Flowchart for Echo

2.3 Exception Handlers

The Monitor441 program uses custom exception handlers. They are loaded using the source code:

2.3.1 Bus Error Exception

2.3.1.1 Algorithm and Flowchart

This exception is called whenever a bus error exception occurs. It outputs the SSW, IR, and BA along with a custom string message. The register values are also printed to the screen. The flowchart is shown in Figure 17.

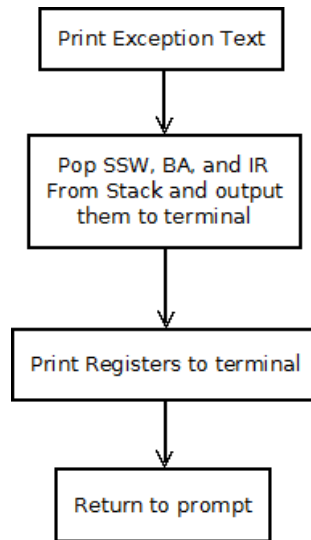


Figure 17: Flowchart for Bus Error Exception

2.3.1.2 Assembly Code

2.3.2 Address Error Exception

2.3.2.1 Algorithm and Flowchart

This exception is called whenever an address error exception occurs. It outputs the SSW, IR, and BA along with a custom string message. The register values are also printed to the screen. The flowchart is shown in Figure 18.

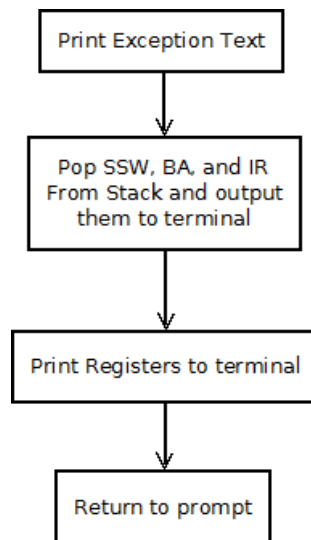


Figure 18: Flowchart for Address Error Exception

2.3.2.2 Assembly Code

2.3.3 Illegal Instruction Error Exception

2.3.3.1 Algorithm and Flowchart

This exception is called whenever an illegal instruction error exception occurs. It outputs a custom string message, and the register values are also printed to the screen. The flowchart is shown in Figure 19.

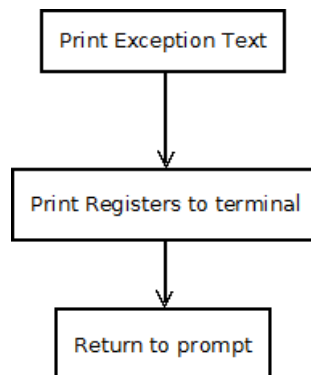


Figure 19: Flowchart for Illegal Instruction Exception

2.3.3.2 Assembly Code

2.3.4 Privilege Violation Error Exception

2.3.4.1 Algorithm and Flowchart

This exception is called whenever an privilege violation error exception occurs. It outputs a custom string message, and the register values are also printed to the screen. The flowchart is shown in Figure 20.

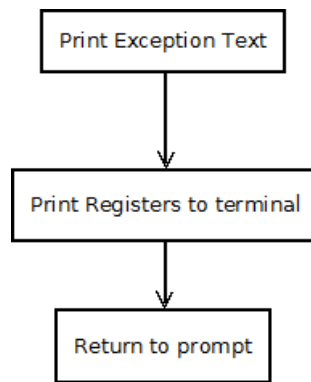


Figure 20: Flowchart for Privilege Violation Exception

2.3.4.2 Assembly Code

2.3.5 Divide by Zero Error Exception

2.3.5.1 Algorithm and Flowchart

This exception is called whenever a divide by zero error exception occurs. It outputs a custom string message, and the register values are also printed to the screen. The flowchart is shown in Figure 21.

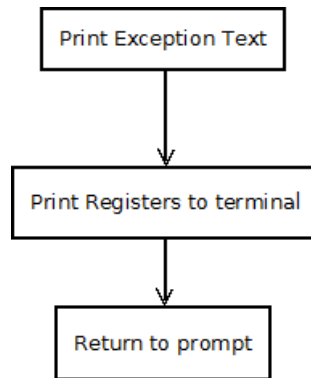


Figure 21: Flowchart for Divide by Zero Exception

2.3.5.2 Assembly Code

2.3.6 A Line Emulator Error Exception

2.3.6.1 Algorithm and Flowchart

This exception is called whenever an A line emulator error exception occurs. It outputs a custom string message, and the register values are also printed to the screen. The flowchart is shown in Figure 22.

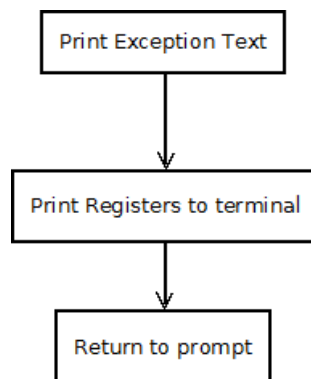


Figure 22: Flowchart for A Line Emulator Error Exception

2.3.6.2 Assembly Code

2.3.7 F Line Emulator Error Exception

2.3.7.1 Algorithm and Flowchart

This exception is called whenever an F line emulator error exception occurs. It outputs a custom string message, and the register values are also printed to the screen. The flowchart is shown in Figure 23.

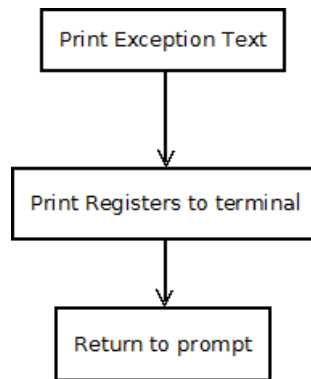


Figure 23: Flowchart for F Line Emulator Error Exception

2.3.7.2 Assembly Code

2.3.8 Check Instruction Error Exception

2.3.8.1 Algorithm and Flowchart

This exception is called whenever a check instruction error exception occurs. It outputs a custom string message, and the register values are also printed to the screen. The flowchart is shown in Figure 24.

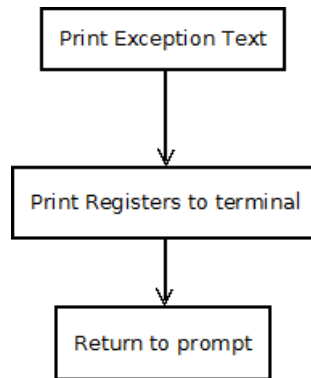


Figure 24: Flowchart for Check Instruction Error Exception

2.3.8.2 Assembly Code

2.4 User Instruction Manual Exception Handlers

2.4.1 Syntax/Unknown Command Error

2.4.1.1 Algorithm and Flowchart

This error is meant to guide the user to input the correct syntax for a command. It first checks if the command entered is valid. If not, an unknown command message is displayed. If the command entered is valid but the syntax is incorrect, an incorrect syntax message is outputted to the terminal. The flowchart is shown in Figure 25.

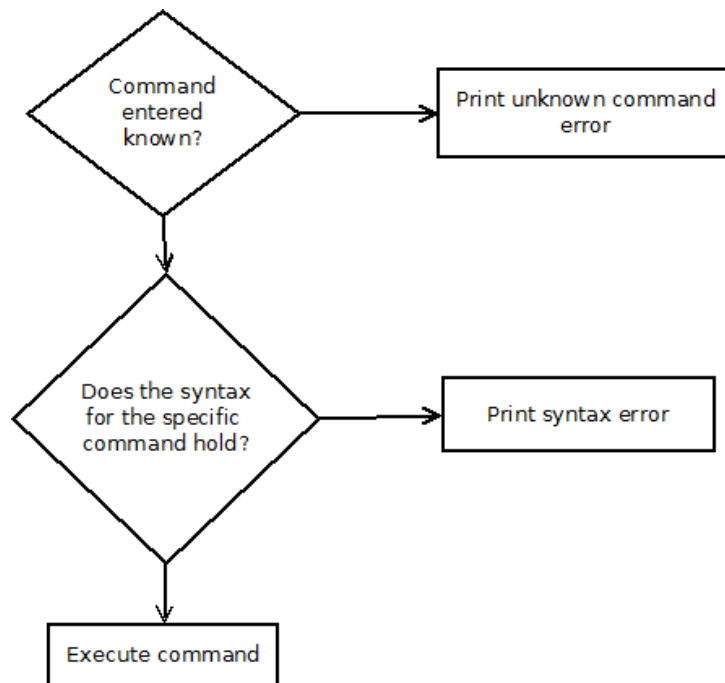


Figure 25: Flowchart for User Instruction Manual Exception Handler

2.4.1.2 Assembly Code

3 Discussion

With a low level programming language such as assembly, the computer engineer is in full control of how each command is built. When building a monitor type program, assembly can be great because of its simplicity, but it also has its draw backs due to a lack of high level API. This increases the chances that erroneous input will kill the program or produce unpredictable results. This means that in order to have a flawless program, the amount of source code needed to be written per command could easily violate the design constraint of having under 3K code size. Because of this, major errors were checked, but it is still possible to break the program by producing minor errors. The program assumes the user knows how to accurately use the tools provided, and if wrong input is entered and an error message is not displayed,

the user will have already known their input was invalid. Furthermore, there are some major limitations to the functionality of this program, for example, certain commands can only take in byte sized data or a hexadecimal to decimal conversion can only accept a maximum value of FFFF. Regardless, this program performs on par with Motorola's Tutor software and, with error checking aside, could replace Tutor entirely.

There were many engineering and design challenges encountered during the journey to construct this program. A majority of these challenges came up during the debugging of each separate command. Taking an algorithm and knowing how to code it is a simple process, but the actual implementation is the hard part. This is when things such as runtime errors must be eliminated. Overall, runtime errors accounted for 95% of the total generated errors, and the code had to be run step by step to pinpoint the exact moment of error in order to be fixed. Furthermore, deciding on how to store and manipulate data was a huge design concern. With only 7 data registers and 7 address registers, storage "containers" had to be carefully picked so that there were no memory leaks and registers needed to be untouched due to subroutines were not accessed accidentally.

4 Feature Suggestions

As discussed earlier in Section 3, complex error checking should be implemented in this program. Furthermore, more commands should be implemented to emulate not just Motorola's Tutor software, but current Operating System distributions as well. This could include commands such as `ls` (listing files in current directory) or `cd` (change directory). If implementing an embedded system using the MC68000 processor, and the `MONITOR441` program is used as a basis, these commands could help the programmer easily analyze top layer applications such as installed files as well as low layer applications such as displaying register values.

5 Conclusion

Overall, the monitor program was created, and it has all of the requested functionality implemented. While it is not 100% error free, it provides the user a great MC6000 based piece of software for use with debugging the

microprocessor. Further work could be done to improve the functionality to be on the level of Motorola's Tutor software, but this shouldn't be done as no modern day technology runs based on the MC68000 microprocessor.

References

- [1] Harman, Thomas L., and Barbara Lawson. *The Motorola MC68000 Microprocessor Family: Assembly Language, Interface Design, and System Design*. Englewood Cliffs, NJ: Prentice-Hall, 1985. Print.
- [2] MC68000 Microprocessor Programmer's Reference Manual
- [3] SANPER-1 Lab Manuals
- [4] MC68000 Educational Computer Board User's Manual