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

List of Figures

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. 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 that 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.

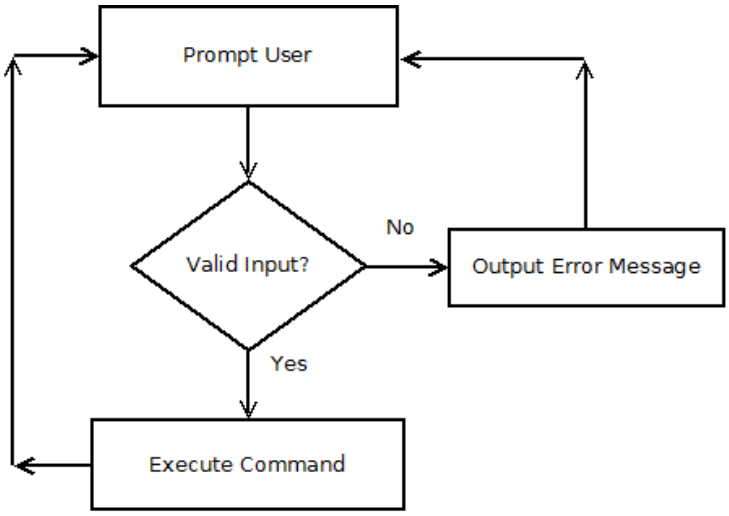

Figure 1: Structure of Monitor Program

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.

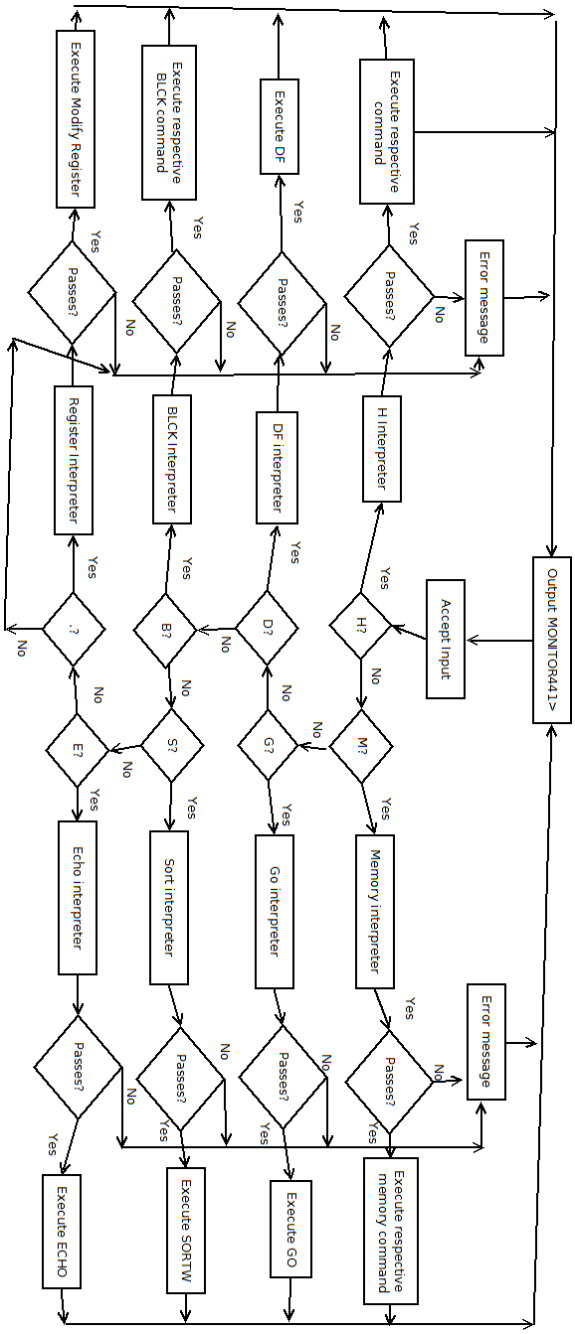

Figure 2: Flowchart for Command Line Interpreter

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.

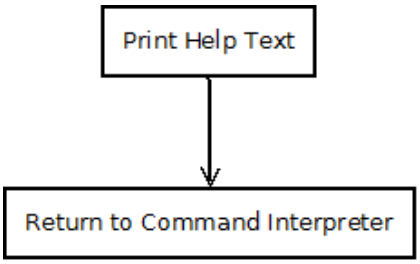

Figure 3: Flowchart for Help

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.

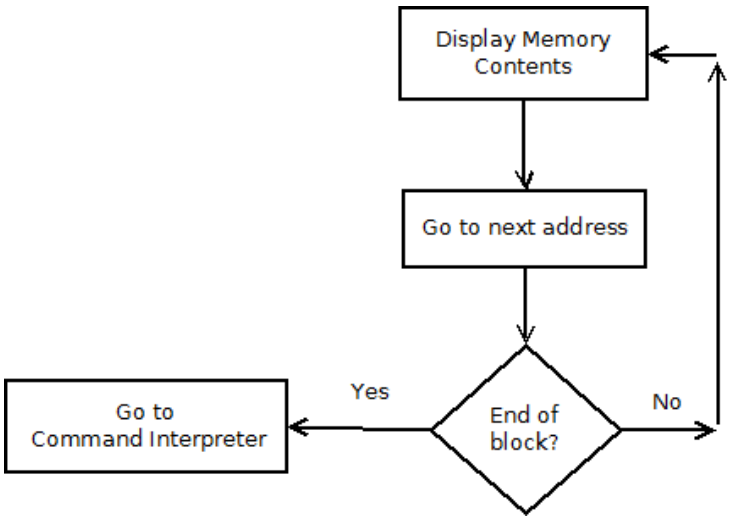

Figure 4: Flowchart for Memory Display

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.

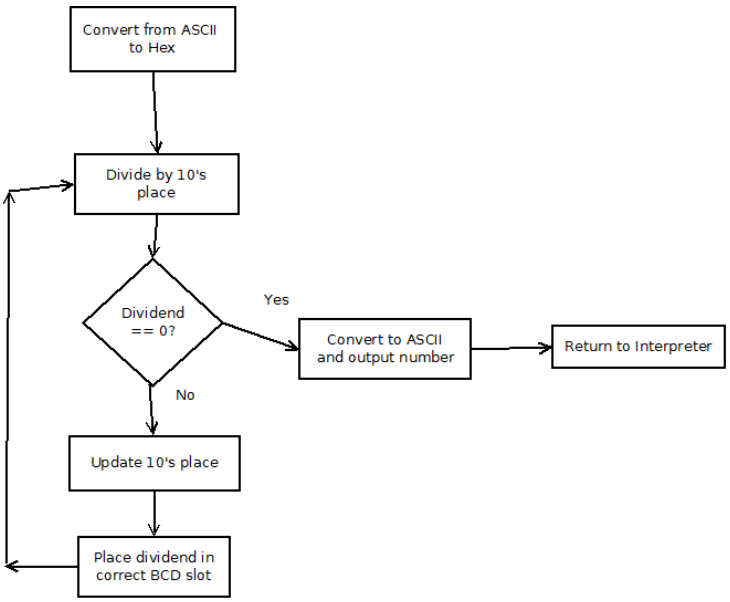

Figure 5: Flowchart for HXDEC

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.

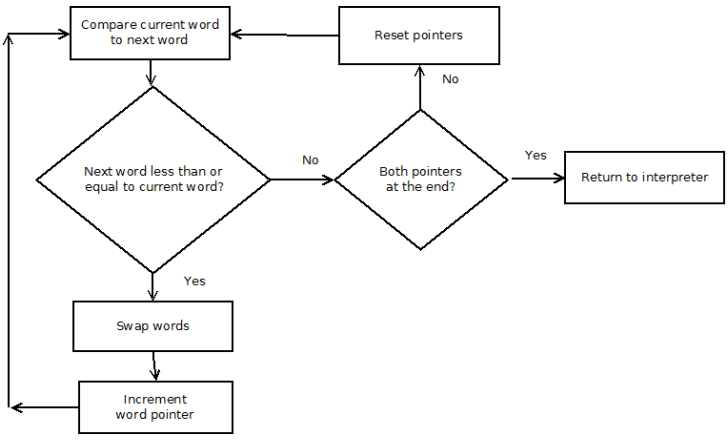

Figure 6: Flowchart for SORTW

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.

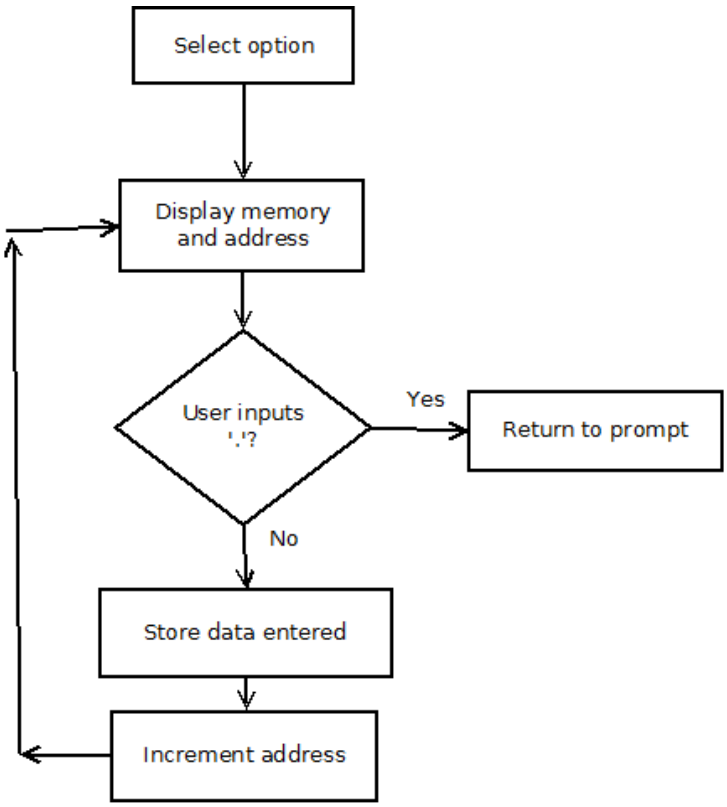

Figure 7: Flowchart for Memory Modify

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 is Figure 8.

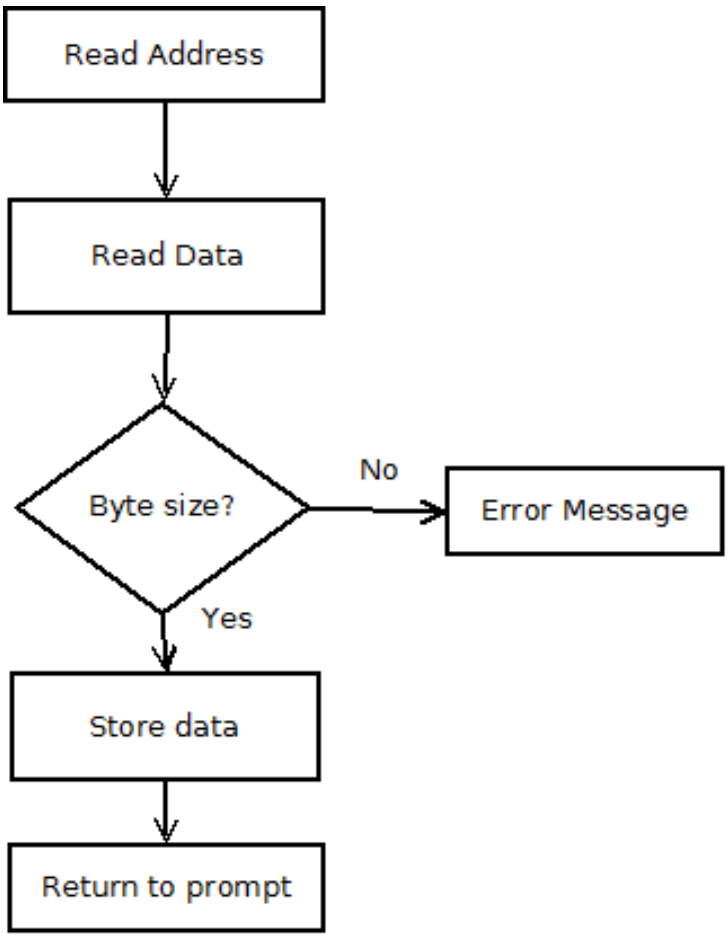

Figure 8: Flowchart for Memory Set

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.

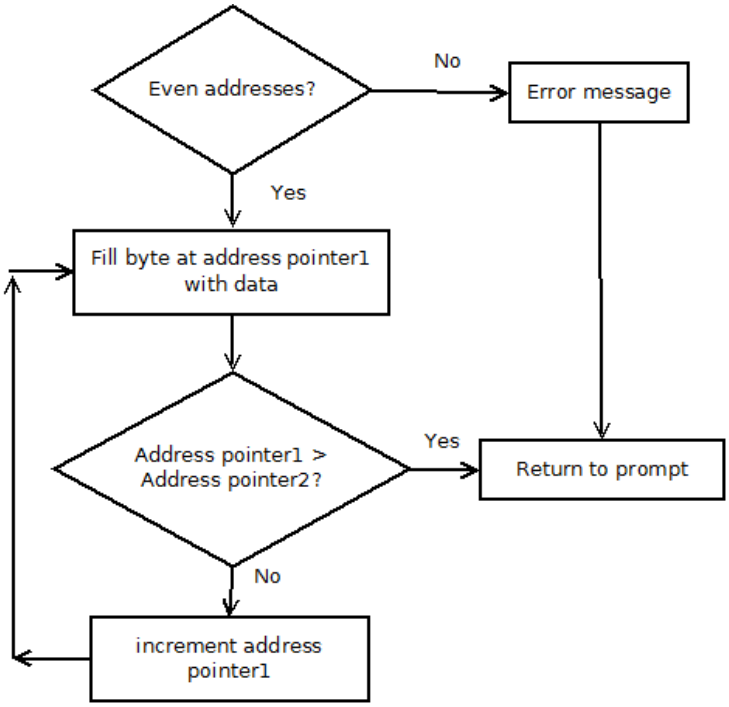

Figure 9: Flowchart for Block Fill

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.

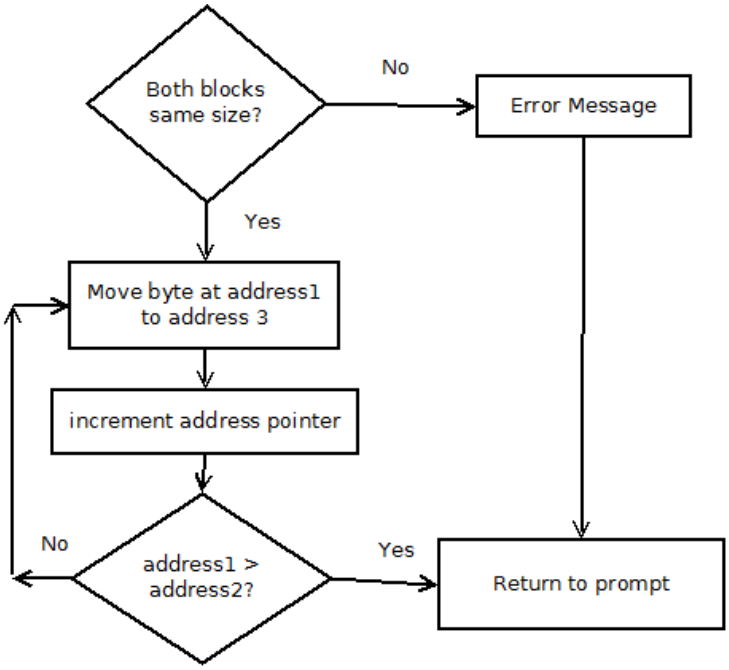

Figure 10: Flowchart for Block Move

2.2.8.2 Assembly Code

2.2.9 Block Test

2.2.9.1 Algorithm and Flowchart

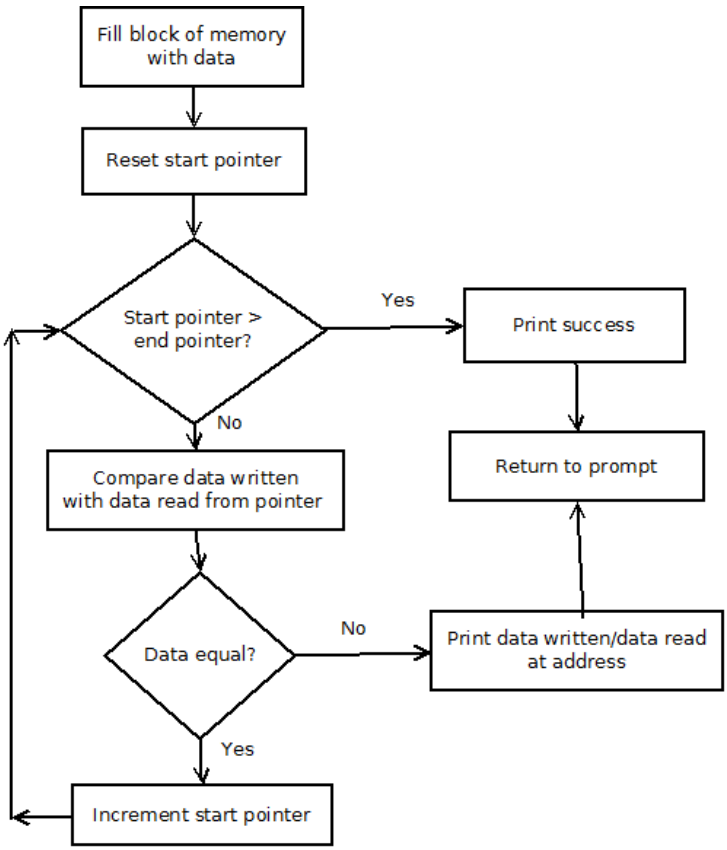

Figure 11: Flowchart for Block Test

2.2.9.2 Assembly Code

2.2.10 Block Search

2.2.10.1 Algorithm and Flowchart

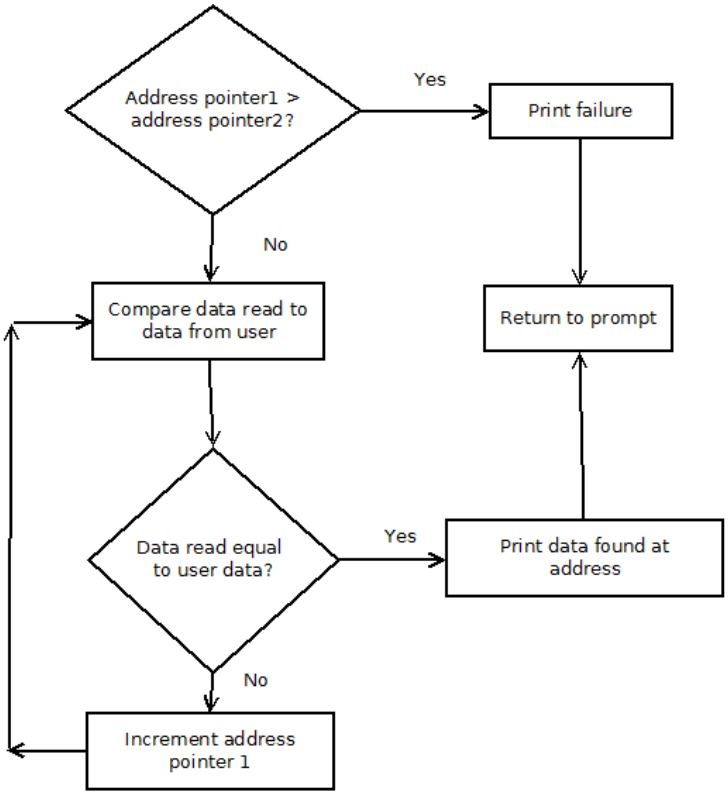

Figure 12: Flowchart for Block Search

2.2.10.2 Assembly Code

2.2.11 Go

2.2.11.1 Algorithm and Flowchart

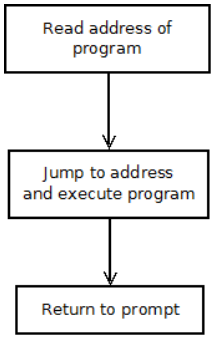

Figure 13: Flowchart for Go

2.2.11.2 Assembly Code

2.2.12 Display Formatted Registers

2.2.12.1 Algorithm and Flowchart

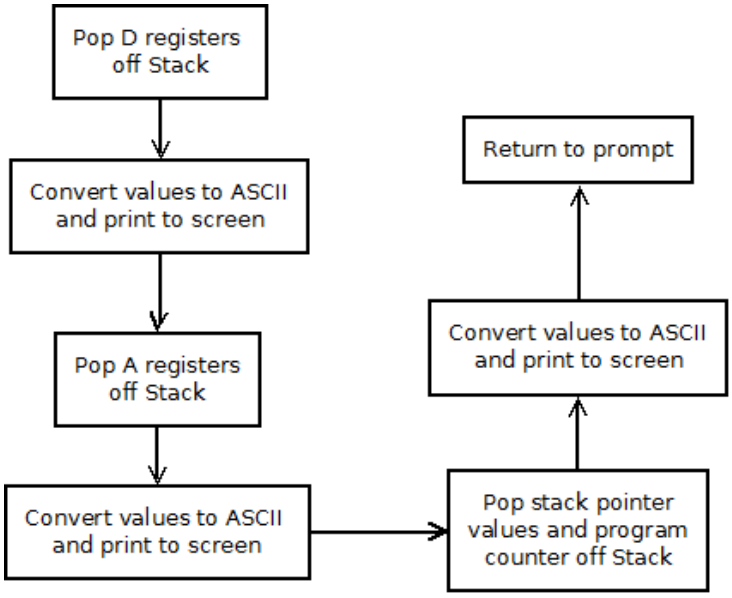

Figure 14: Flowchart for Display Formatted Registers

2.2.12.2 Assembly Code

2.2.13 Modify Register

2.2.13.1 Algorithm and Flowchart

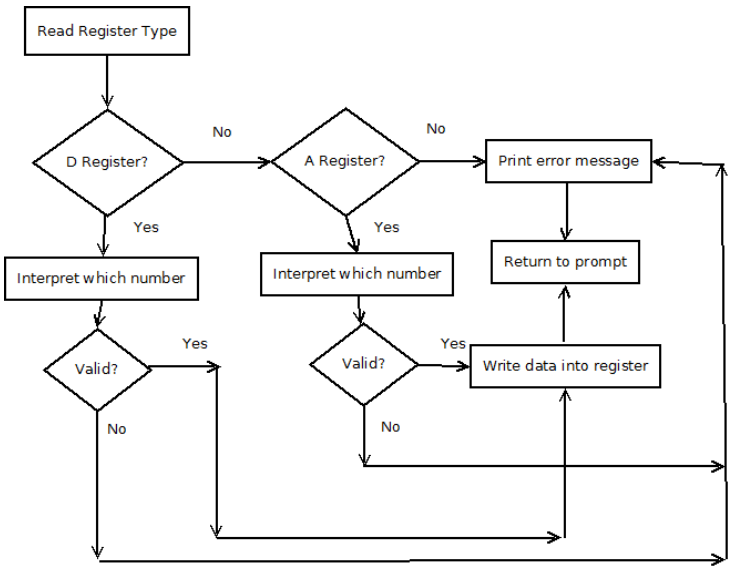

Figure 15: Flowchart for Modify Register

2.2.13.2 Assembly Code

2.2.14 Echo

2.2.14.1 Algorithm and Flowchart

2.2.14.2 Assembly Code

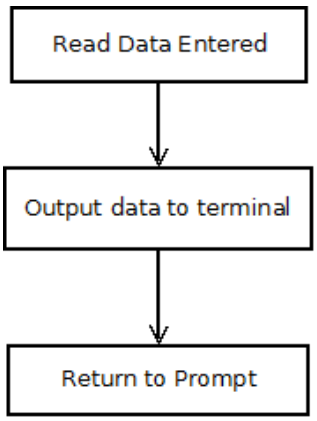

Figure 16: Flowchart for Echo

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

2.3.1.2 Assembly Code

2.3.2 Address Error Exception

2.3.2.1 Algorithm and Flowchart

2.3.2.2 Assembly Code

2.3.3 Illegal Instruction Error Exception

2.3.3.1 Algorithm and Flowchart

2.3.3.2 Assembly Code

2.3.4 Privilege Violation Error Exception

2.3.4.1 Algorithm and Flowchart

2.3.4.2 Assembly Code

2.3.5 Divide by Zero Error Exception

2.3.5.1 Algorithm and Flowchart

2.3.5.2 Assembly Code

2.3.6 A Line Emulator Error Exception

2.3.6.1 Algorithm and Flowchart

2.3.6.2 Assembly Code

2.3.7 F Line Emulator Error Exception

2.3.7.1 Algorithm and Flowchart

2.3.7.2 Assembly Code

2.3.8 Check Instruction Error Exception

2.3.8.1 Algorithm and Flowchart

2.3.8.2 Assembly Code

2.4 User Instruction Manual Exception Handlers

2.4.0.3 Algorithm and Flowchart

2.4.0.4 Assembly Code

3 Discussion

With such a low level programming language such as assembly, the computer engineer is in full control

4 Feature Suggestions

5 Conclusion

References

[1] test