ECE 441

Microprocessors

Instructor: Dr. Jafar Saniie

Teaching Assistant: Guojun Yang

Final Project Report:

**MONITOR PROJECT**

09/30/2017

By: Hamza Shakeel

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

Abstract 2

1-) Introduction 2

2-) Monitor Program 3

2.1-) Command Interpreter 4

2.1.1-) Algorithm and Flowchart 4

2.1.2-) 68000 Assembly Code 6

2.2-) Debugger Commands 7

2.2.1-) HelP 7

2.2.2-) MDSP 8

2.2.3-) SORTW 10

2.2.4-) MM 12

2.2.5-) MS 17

2.2.6-) BF 19

2.2.7-) BMOV 20

2.2.8-) BTST 21

2.2.9-) BSCH 23

2.2.10-) GO 25

2.2.11-) DF 26

2.2.12-) EXIT 28

2.2.13-) MOVEA 28

2.2.14-) CMPI 30

2.3-)Exception Handlers 31

2.3.1-) Bus Error Exception 31

2.3.2-) Address Error Exception 33

2.3.3-) Illegal Instruction Exception 33

2.3.4-) Privilege Violation Exception 33

2.3.5-) Divide by Zero Exception 33

2.3.6-) Line A and Line F Emulators 33

3-) Discussion 33

4-) Feature Suggestions 34

6-) Conclusions 34

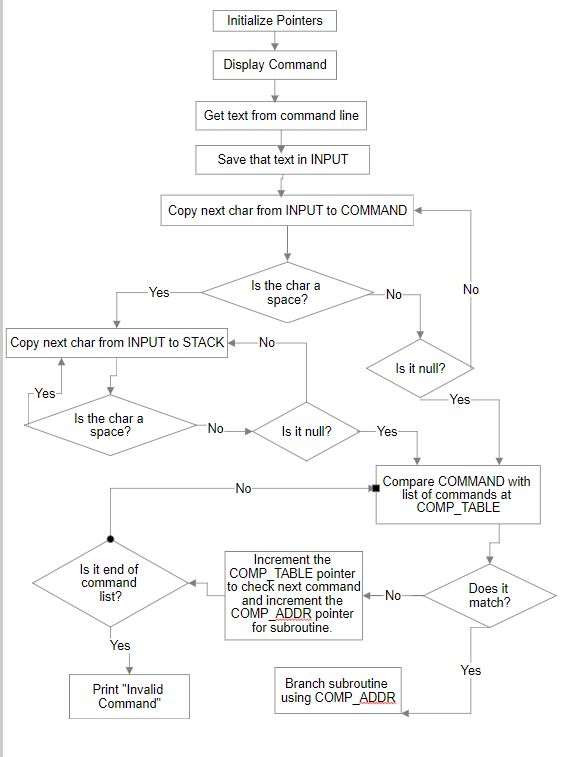
7-) References 34

Abstract

This purpose of this design project is to build a Monitor program with MC68000 assembly language. This Monitor program includes a set of commands that can help the user with debugging and running the programs. The program can also handle exceptions like bus or address exception. This Monitor program was build using MC68000 simulator called Easy68k. The rest of the report will provide more details on how these functions were implemented in the project.

# *1-) Introduction*

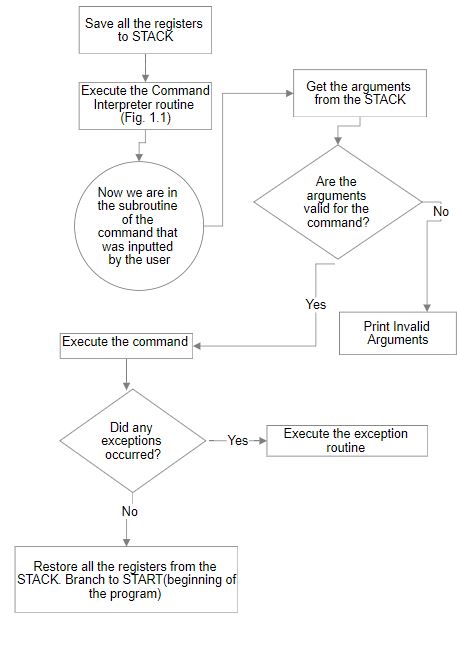
The Monitor program provides a set of debugging commands to the user. These commands allow the user to view and modify the contents of the memory and to run programs that are stored on the memory. The Monitor program will output a prompt when it starts up. The user will then type in one of the commands with the arguments. If the syntax of the command is correct, the Monitor will execute the command, otherwise, it will tell the user that it’s an invalid command. Here is what happens when the Monitor first starts up and user types in a command.



*Figure 1.1. Monitor command interpreter flow chart*

***2-) Monitor Program***

The very first thing the Monitor program will do is save all the registers to the stack so none of the user data is lost. This line of code is labeled START and this is where our Monitor program will start execution. Then, it will initialize all the pointers and execute the command interpreter routine described above in figure 1.1. At the end of the command interpreter routine, we will branch to subroutine of the command that was typed in by the user. Then, we will get the arguments from the stack and we will make sure that the arguments typed are valid for that command. Then the subroutine will get executed. Figure 2.1 describes this whole process. Some of the subroutines are not included in this diagram. A detailed version of those subroutines will be described later in the report.



*Figure 2.1. Monitor program*

***2.1-) Command Interpreter***

The first thing we need to do is interpret the command that the user typed in. First, whatever the user typed in, including the arguments, will be stored in a INPUT buffer. The second step is to store the command in the COMMAND buffer and the arguments on the stack. Since we know that the first word will always be the command and there are no two word commands, we can start copying the first word of INPUT to COMMAND. The algorithm for correctly copying the first word of INPUT to COMMAND is shown below.

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

We will copy one character at a time from INPUT to COMMAND. If the character is not space or null, we will copy the next character until we reached space or null. We will check for both a space and a null for the end of the first word because commands like help will end with a null character because they don’t take any arguments and commands with arguments will end with space followed by the argument. We will also have a counter to make sure our command isn’t more than 8 characters.

*Set counter =9 // set the counter*

*Label Move one char from input to command // to move the first word to command buffer*

*Decrement counter by 1 // to ensure command isn’t > 8 chars*

*Is the char a space? // to check for the end of the first word*

*If yes then add null at the end // add null at the end of command to mark the end*

*Branch to move arguments to stack(MTS)*

*Is the char a null? // to check for the end of first word*

*If yes, branch to compare- // this command doesn’t have any arguments*

*Command(CC) with command table*

*Otherwise branch back to Label*

*MTS move arguments to stack // move arguments to stack if needed*

*CC compare COMMAND with CMP\_TABLE // compare it with list of commands*

*If match, branch to address // found the command. Execute subroutine*

*Specified by COMP\_ADDR*

*Otherwise, increment COMP\_TABLE // to compare it with next entry*

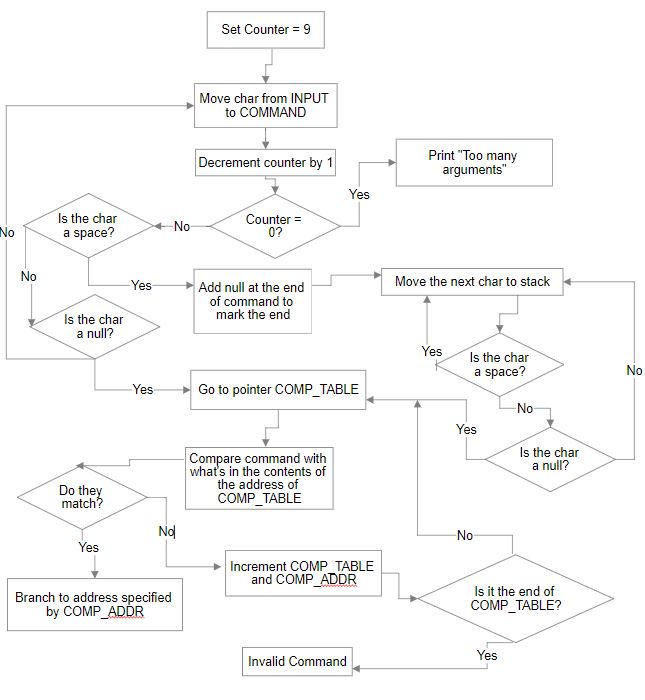
*And increment COMP\_ADDR*

*Have we reached the end of CMP\_TABLE?*

*If yes, then the command is invalid*

*Otherwise, branch back to CC // to compare with next command*

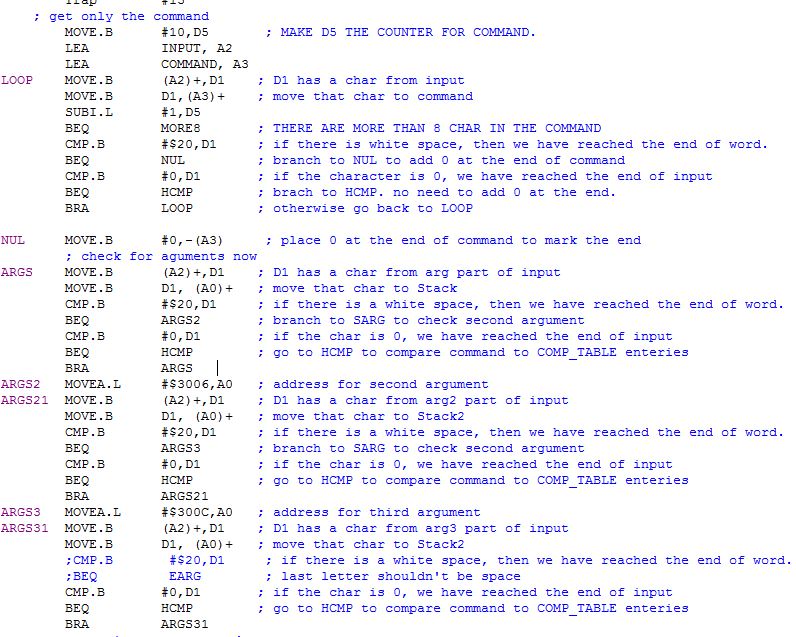
*Figure 2.2. Command Interpreter Algorithm*

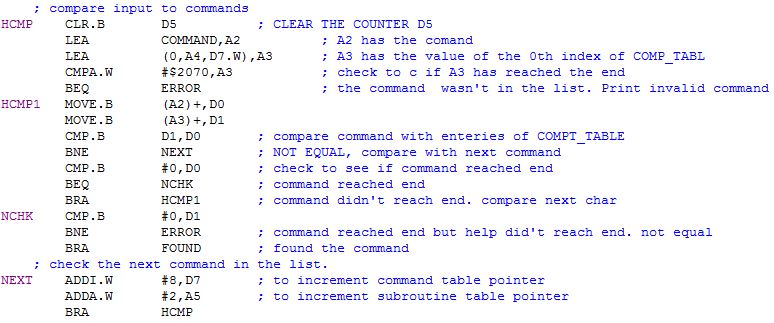


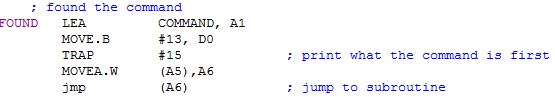
***2.1-)***

*Figure 2.3. Command Interpreter Flowchart*

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







*Figure 2.4. 68000 Assembly Code*

***2.2-) Debugger Commands***

There are 14 debugger commands that I implemented. One of these commands is a HELP command that describes the functions of each command and their syntax. These commands help the user view and change the contents of the memory and some registers. The user can also execute their own programs using the GO command. The user can search through or sort the contents of the memory too. A detailed description of the implementation of each of these commands is described below.

***2.2.1-) HELP***

The first command is a simple HELP command. This command will list the names of all the command that are available to the user, their syntax, and their functions.

***2.2.1.1-) HELP Algorithm***

To implement this command, we simply printed the name of the command followed by the description of the command and an example of how to use it that shows the syntax of the command as well.

*Command 1 name: A brief description of what the command does and how it should be used. Ex: COMAND\_NAME1 Arg1 Arg2*

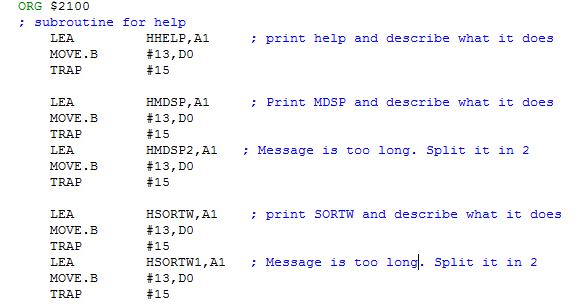
*Command 2 name: A brief description of what the command does and how it should be used. Ex: COMAND\_NAME2 Arg1 Arg2*

*….continue for all 14 commands*

*Figure 2.5. Debugger Command # 1 Algorithm*

***2.2.1.2-) HELP Assembly Code***

Here is the assembly code for HELP. The following code just prints whatever is stored in the labels that are being moved into A1 register. The labels are defined somewhere else. For example, LEA HHELP, A1 will move the address of HHELP in A1 and the string stored there will be printed. HHELP just prints out “HELP: Description of help. Syntax of HELP.” The same code is then executed for rest of the commands. The whole code isn’t shown because of similarities between the rest of the code and the portion that is shown.



*Figure 2.6. HELP Assembly Code*

***2.2.2-) MDSP***

MDSP is a command that displays the contents of memory from address 1 to address 2. These addresses are passed as arguments. If there is only one argument passed, it will print contents of memory from address 1 to (address 1 + 16 bytes).

***2.2.2.1-) MDSP Algorithm and Flowchart***

First, we will check to make sure that the user passed in valid arguments for MDSP. MDSP can have either one address as argument or two addresses. Here is the algorithm for checking the arguments and then storing the arguments in A registers. This algorithm will be repeated, with slight modifications, to check for arguments for other commands as well. One thing to note is that the algorithm uses a subroutine called ATOHEX that converts the ASCII value from A0, my STACK, to hex value. The result is saved in stack.

*Check the address of stack // the location of stack determines the*

*If stack = $3006, one arg typed // number of arguments*

*Go to label ONEARG // Makes Arg2=(Arg1+16 bytes). Then comes back here.*

*If stack > $300C // too many args were typed*

*Invalid argument*

*get the first ARG from A0 // get the ASCII value of ARG1 from stack*

*BSR ATOHEX // Change ASCII to HEX. Result in stack*

*Move from stack to A2 // first address is now saved at A2*

*Get the 2nd address from*

*BSR ATOHEX*

*Move from stack to A3 // second address is now at A3*

*Sub A3-A2 // Second address should be bigger*

*BLT EARG // if not, branch to EARG(error in argument)*

*Rest of the code………………*

*Figure 2.7. Checking arguments algorithm*

Now that we have verified that the arguments are valid and we have successfully saved the two addresses in the A registers, we can start the MDSP command.

*DISP Print A2 // print the first address*

*Print space and then memory contents*

*Go to next line*

*Add one to A2*

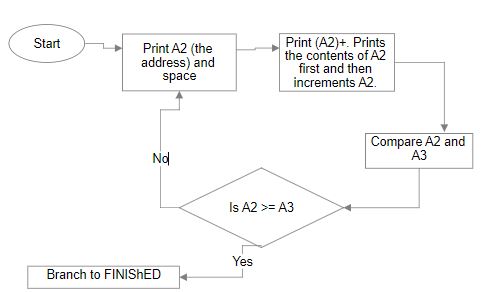
*CMP A2 and A3 // to check if we have reached the end?*

*BGE DISP // if A3>=A2, branch to DISP*

*BRA FINISHED // otherwise we are done, branch to finished*

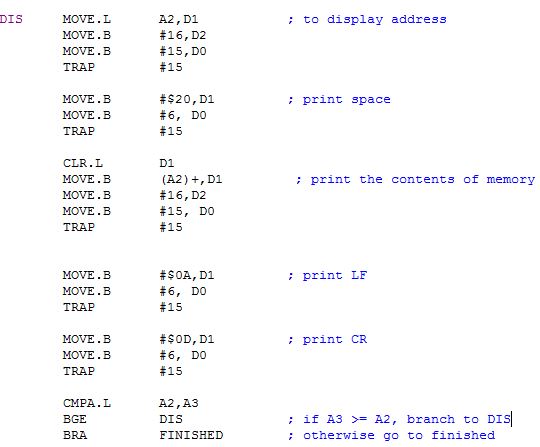
*Figure 2.8. MDSP algorithm*

The above command will print the address and the contents of memory in that address. It will increment the first address and do this again until we reached the second address. Once it reaches the end, we will branch to FINISHED label. This label will simply restore the registers from the stack so no data is lost and then it will branch back to START label (the beginning of the Monitor program). The flow chart below will help better understand the above algorithm.



*Figure 2.9. MDSP flowchart*

***2.2.2.2-) MDSP Assembly Code***



*Figure 2.10. MDSP assembly code*

The above code is based on the algorithm described in fig 2.8. This prints the address, then space, then the contents of memory at that address. The LF and CR were added to make sure everything isn’t

***2.2.3-) SORTW***

The next command is SORTW or sort word. This sorts a block of memory starting from address 1 to address 2. The third argument describes if the user wants it sorted in ascending order or descending. The syntax looks like this: SORTW $add1 $add2 A. First step is same as the last time, to make sure that the number of arguments, along with the arguments themselves, are correct. The algorithm and code for this step is very similar to the last step, so we will not go over it. We also used the same ATOHEX subroutine to convert the ASCII values of the address to hex so we can use the address to do arithmetic.

***2.2.3.1-) SORTW Algorithm and Flowchart***

I used a sorting algorithm called bubble sort. This compares 2 values at a time and swaps them so they end up in the right place. Fig 2.11 describes the algorithm in detail. This description is to sort in ascending order. At his point, the 1st address is in A2, while 2nd address is in A3. A4 also has a copy of first address coz A2 will be modified. D4 will be used as a counter.

*Subtract D4=A3-A2 // to compare the 2 addresses. D4 has the block size*

*If negative result, EARG // A2 shouldn’t be bigger, invalid Args*

*If D4 <4, EARG // the block size is too small to sort word*

*ASC MOVE.W (A2)+, D2 // D2 has the first value, A2 is incremented*

*MOVE.W (A2), D3 // D3 has the next value*

*Compare D2, D3*

*If D2 is bigger*

*Swap the 2 values // so bigger value will end up on the right,*

*Compare A2 and A3 // otherwise leave them alone*

*If A2=A3, branch to RO // first round over.*

*Otherwise, brach to ASC // compare next to*

*RO D4 = D4-2 // decrement counter by 2 coz word size*

*D4=0? If yes, branch to FINISHED // we have reached the end*

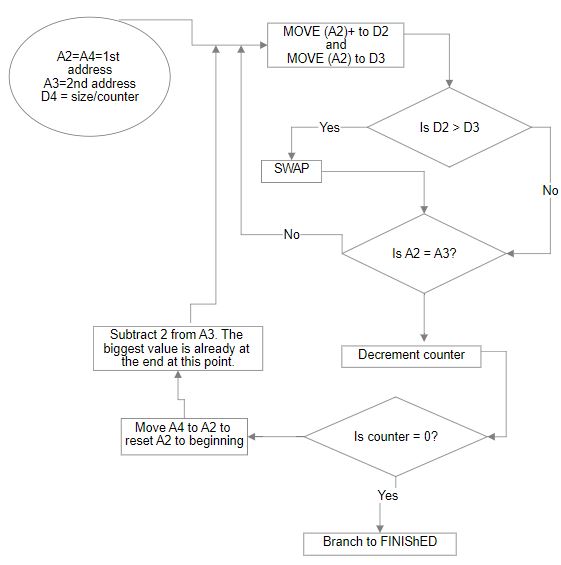
*Move A4 to A2 // A4 had a copy of first address. A2 is reset.*

*A3=A3-2 // decrement the second address coz the biggest // value is already in the last place.*

*BRA ASC // branch to compare next 2 values*

*Figure 2.11 SORTW algorithm*

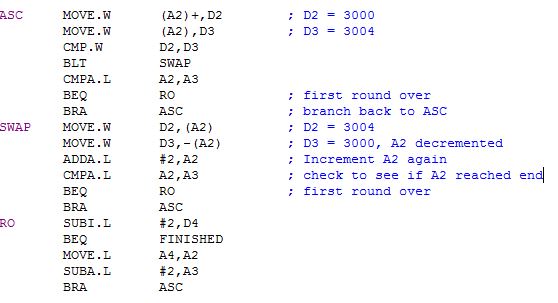
This algorithm is shown again on the flowchart. The flowchart will help with the visualization.



*Figure 2.12 SORTW flowchart*

The flowchart above show the algorithm for sorting the memory in ascending order. This algorithm will compare the first 2 values, and move the bigger value to the right. Then it will compare the next two values, and again will move the bigger value to the right. It will do the same thing for all the values until it reached the end of the list. At this point, the largest value will be placed at the end of the list. This marks the end of the first round. After this, we will move the end of the list to the second last value by decrementing the counter. This is done because the last value is already in the right place so we don’t need to look at it. Then, start this process all over again. We will keep doing this process until the counter reaches 0. This will sort our list in ascending order. To sort it in descending order, all we have to change is when to swap. We will swap if the D2 is less than D3 instead of swapping when it’s bigger.

***2.2.3.2-) SORTW Assembly Code***



*Figure 2.13 SORTW assembly code*

In order to avoid repeating code, the code for argument checking and to sort in descending order has been omitted.

***2.2.4-) MM***

The next command is memory modify(MM), this command displays the contents of memory and modifies it if needed. It can modify a byte, word, or a long word of memory. The syntax for this is: MM $4000 ;B. This will display the contents of memory in bytes at address $4000. If user wants to modify it, he can enter two characters and those characters will replace the contents at $4000. Then the next address will be displayed along with the memory contents. User can modify that too or he could skip it by pressing enter. To return to the prompt, the user will type “.” without the quotation and hit enter.

***2.2.4.1-) MM Algorithm and Flowchart***

Just like the last command, we will skip the argument checking part of this command. At this point, we already have the address saved in stack along with: ;B, ;W or ;L argument. We will also make use of a subroutine A2HEX that will convert 2 bytes of ASCII data to hex. This routine is different from previously used ATOHEX routine, which converted 4 ASCII characters to 4 hex numbers.

*Put second argument in D7 // so we can compare it with B, W, or L*

*Is it B? if yes go to B*

*Is it W? if yes go to W*

*Is it L? if yes go to L*

*B moves 1 in D7 // D7 will let us know how many times to run*

*W move 2 in D7 // A2HEX subroutine*

*L move 3 in D7*

*Move D7 to D4 // make a copy of D4*

*loop Print the address(A0)*

*Print the contents // prints only 2 bytes*

*D4-1 // this will print 2 bytes if a byte*

*If D4==0 // will print 4 bytes if a word, 6 if a long*

*BRA exit // get out of the loop*

*Bra loop*

*Exit print space and other stuff to make the output look nicer*

*Read string from keyboard. Stored in A1*

*Makes sure it has appropriate number of characters*

*Compare (A1) with CR*

*If CR, skip by incrementing the address and branching back to loop*

*If . branch to FINISHED*

*AG Jump to A2HEX to convert 2 ASCII bytes to Hex*

*Update memory*

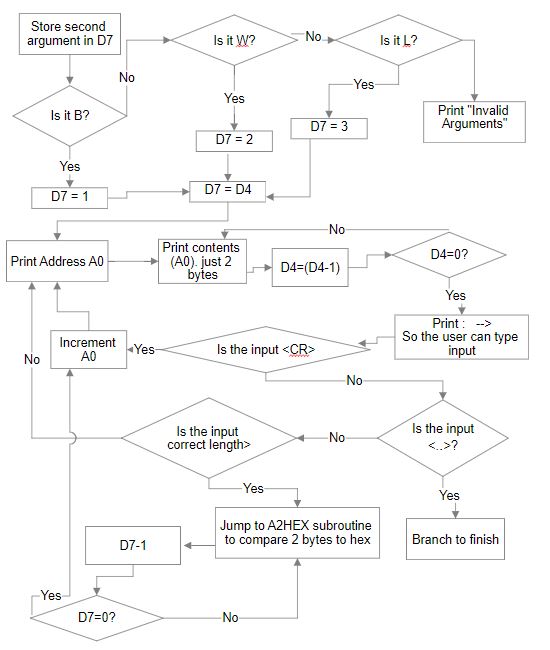
*D7-1 // for byte, after printing 2 letters, we’ll branch to loop*

*If D7=0, branch to loop // for word, after 2 iterations, we’ll will branch to loop*

*BRA AG // for long, after 3 iterations, we’ll branch to loop*

*Figure 2.14. MM algorithm*

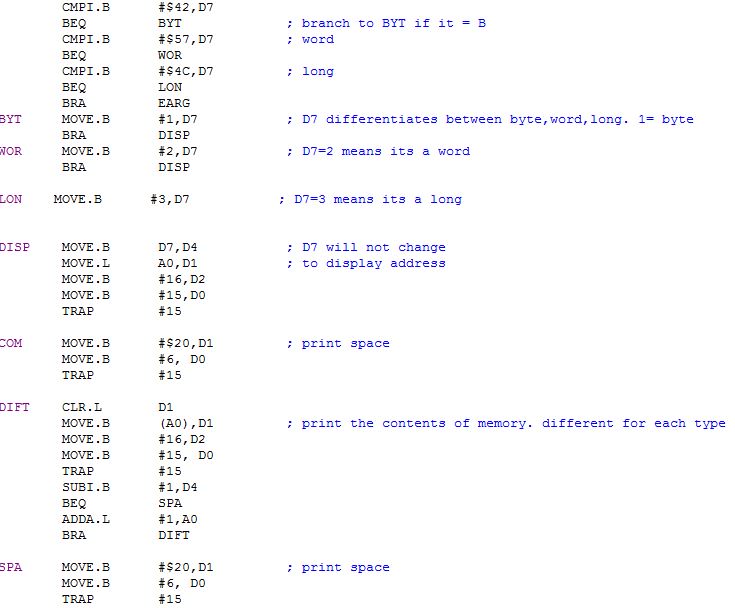
The above algorithm works for byte, word, and long, but there is a mistake in the algorithm. It only prints 3 bytes for the long instead of printing 4 bytes. The algorithm is pretty simple, but the implementation is very complicated. I realized my mistake of assigning 3 to D7 for long after I already finished writing the code. To go back and change it would require a lot of code modifications and could essentially mess up a working method. So, I decided to leave it like that instead of fixing it. This method used a lot more logic but reduced the number of lines of code significantly as I didn’t have to write the same code 3 times. The flowchart below attempts to explain the algorithm better. The flowchart shows how many decisions I had to make in the implementation of this algorithm.



*Figure 2.15. MM flowchart*

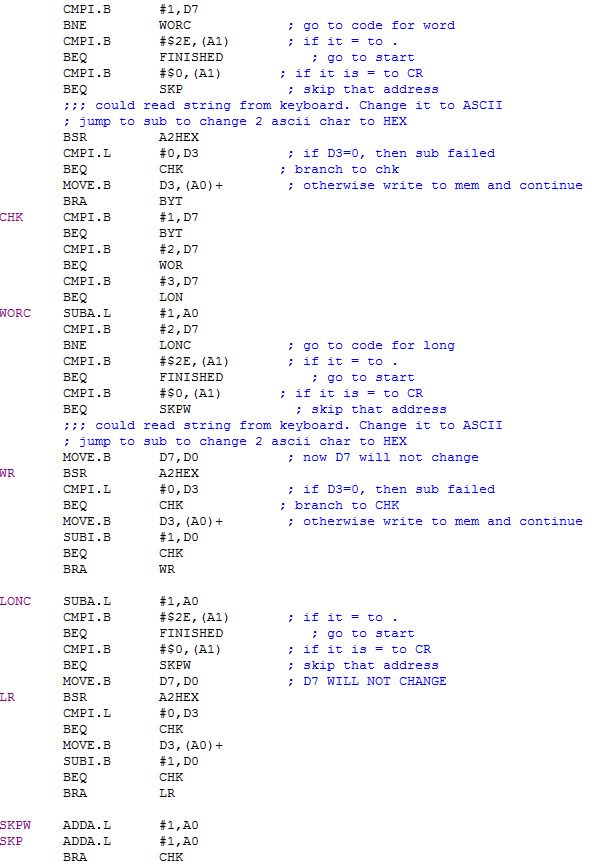
***2.2.4.2-) MM Assembly Code***

Since MM can modify byte, word, or long data, it was the hardest command to implement. This also means that it has the most lines of code. To avoid printing pages of code for this command, I have only included the lines of code that are essential to the main algorithm and just explained in word what the rest of the code does. First, we check the arguments like we did the previous times, If the arguments are correct, we will store the second argument in D7. Then, the next part is shown in the code below.



*Figure 2.16.1 MM Assemble code*

The above code puts the right values in D7 based on if the user typed B, W, or L. Then it makes a copy of D7 and puts in D4 so D7 doesn’t get modified. Then the address of A0, gets printed. This is the address that user typed in for argument 1. Then we print space to make it look nice. Then the contents get printed 2 bytes at a time starting with the label DIFT. The number of time we go back to the DRIFT label depends on if it’s a byte, word, or long. After printing the contents of memory, we print space again. The next few lines of code are not shown since they just print 🡪 and space to make it look nice. Then we take an input from the keyboard. The code that is shown below is after reading the input from the keyboard.



*Figure 2.16.2 MM Assemble code*

The code above first checks the input. If the user just pressed enter, it will branch to skip. If the user typed . then it will branch to FINISHED. If none of these are true, it will convert what user typed in from ASCII to HEX two bytes at a time. It will then decrement D7 and type the next two bytes depending on if it’s a word or a long. This could be better understood by following the flowchart. If the user didn’t type in the right data, it will print the same address and its memory contents and it will ask the user to type the data again. If the user typed the correct data, it will save that data and ask print the next address along with its memory contents.

***2.2.5-) MS***

The next command is memory set (MS). It alters memory at the address specified by the user. The data could take a form of ASCII or hex. The syntax for MS is: MS $4000 hello. This will save hello in ASCII. To save a hex value, do this: MS $4000 $12. It will save hex 12 in the memory. It can only save one byte of data in hex.

***2.2.5.1-) MS Algorithm and Flowchart***

Like any other command, we will first get the arguments. The first argument is the address and it will be stored in A2. The second argument is the data and that will be stored in address specified by A3.

*Move the first char (A3) in D2*

*Is D2 = $ // to check if it’s a hex data*

*If yes, branch to hex // yes branch to hex*

*STRI Move (A3)+,D2 // move one byte at a time*

*MOVE D2,(A2)+ // data is in ASCII form*

*Check if D2 is null // if yes, we have reached the end*

*If yes, branch to finished*

*Else, branch to STRI*

*HEX SKIP the first char // because it’s a $ sign*

*Put the next two bytes in (A1) // the next 2 bytes are what we need to convert*

*Convert it to hex*

*Save it in (A2)+ // A2 has the address where we want to save*

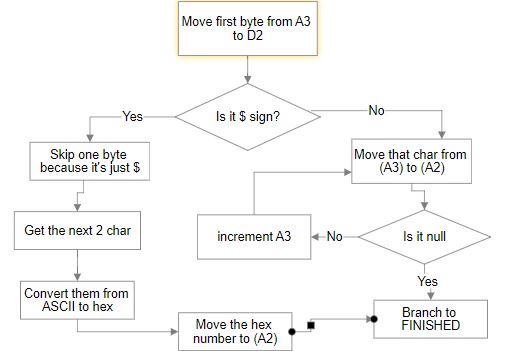
*Is it nul // we are done*

*If yes, branch to finish*

*Otherwise branch to hex*

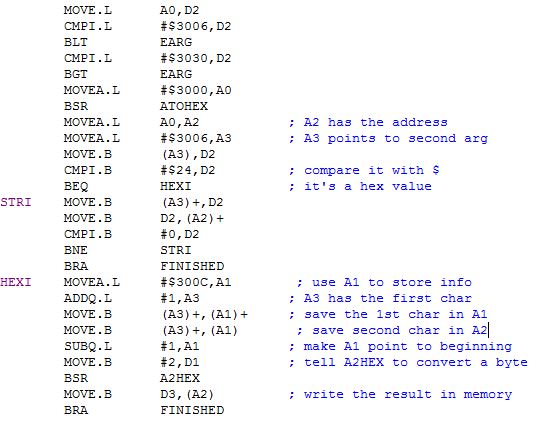
*Figure 2.17. MS algorithm*

The algorithm for MS is pretty simple. I already have a subroutine that converts a byte from ascii to hex so I can use that subroutine for MS to store a hex. To store a ASCII is simple because we are just storing from one location to another. The flow chart is also very simple and it is shown below.



*Figure 2.18. MS flowchart*

***2.2.5.2-) MS Assembly Code***



*Figure 2.19. MS assembly code*

The code on the top starts of by checking the arguments by looking at the stack position. Then it gets the argument 1 from stack and converts it to a hex number. This is where we will store the result. Then it looks at the first char of the second argument to determine if it is ASCII or Hex. Then it just does the same thing that is shown in the algorithm and the flowchart.

***2.2.6-) BF***

The next command fills a word block of memory. It takes in 3 arguments. The first two are the addresses that make up the block. And the last one is data. The syntax looks like this: $ADD1 $ADD2 1234.

***2.2.6.1-) BF Algorithm and Flowchart***

*BEG Make sure the first address isn’t bigger // the range is from ADD1 to ADD2*

*If it is, then branch to EARG // invalid argument*

*Make sure the two addresses aren’t odd // word can’t be in odd address*

*Make D0 = 2 // that’s the counter*

*TH Change a byte from ASCII to hex*

*Move that hex to memory // to fill the block*

*Increment ADD1*

*Decrement counter*

*D0=0? If no, branch to Th // fill the next byte*

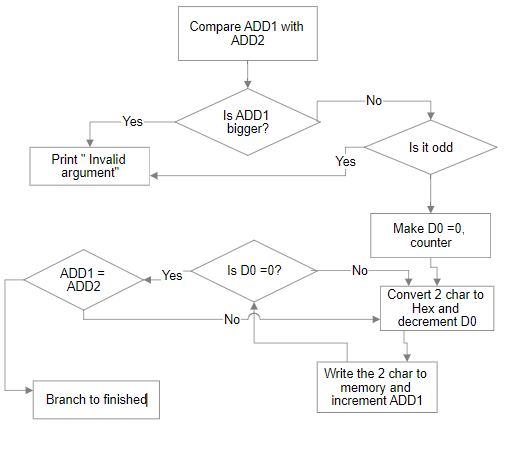
*CHK IS ADD1 = ADD2? // to check if we reached the end.*

*If no branch to BEG*

*Otherwise branch to FINISHED*

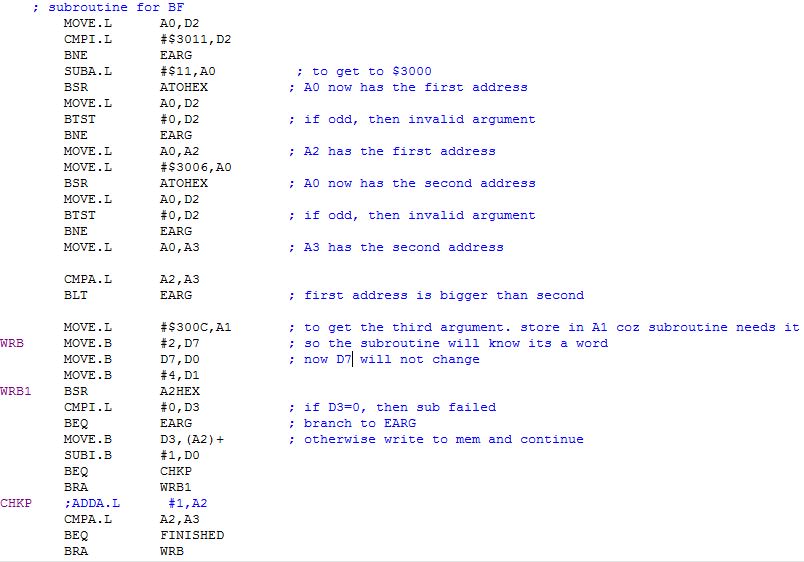
*Figure 2.20 BF algorithm*

This command is very simple, it just puts whatever the user types in the memory. If then it increments the address and checks to see if we reached the end. If we haven’t, it keeps putting same thing in the next memory location until it reaches the end. Here is the flowchart for this.



*Figure 2.21 BF flowchart*

***2.2.6.2-) BF Assembly Code***



*Figure 2.22 BF assembly code*

***2.2.7-) BMOV***

The next command moves a block of memory from one location to another. This command takes in 3 arguments. The first 2 arguments are the range of the data that you need to move. The third argument is where you need to move it to. So if the user types, BMOV $ADD1 $ADD2 $ADD3, this will move everything from ADD1 to ADD2. It will move it to ADD3.

***2.2.7.1-) BMOV Algorithm and Flowchart***

*Loop Make sure the first address isn’t bigger // the range is from ADD1 to ADD2*

*If it is, then branch to EARG // invalid argument*

*Move what’s in ADD1 to ADD4*

*Add 1 to ADD1 // increment ADD1*

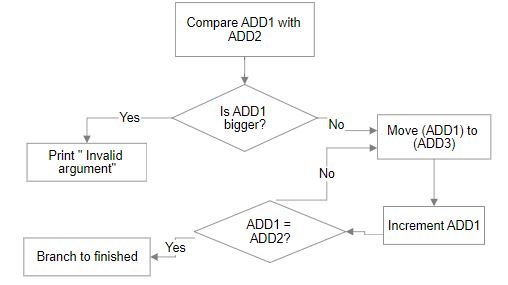
*If ADD1 = ADD3? // check to see if reached end*

*If no branch to loop*

*Otherwise branch to done*

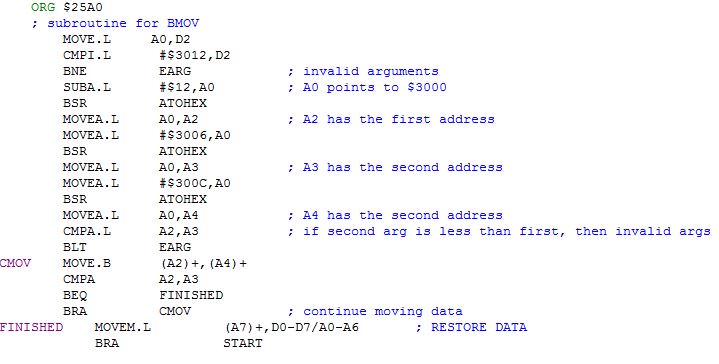
*Figure 2.23 BMOV algorithm*

The flowchart isn’t very complicated for this one. It is just moving data from one address to another. There is no conversion that needs to be done. The argument checking is very similar to the last command that’s why it’s not shown in the flowchart.



*Figure 2.24 BMOV flowchart*

***2.2.7.2-) BMOV Assembly Code***

******

*Figure 2.25 BMOV assembly code*

***2.2.8-) BTST***

This is a destructive test. It tests a block of memory by first writing to it, and then reading to it. If it reads the same value it wrote, the test is passed. If for some reason, it read a different value then it wrote, the test failed. The syntax of the command is like this: BTST $4000 $400A

***2.2.8.1-) BTST Algorithm and Flowchart***

*W Write $AA in the (ADD1) // write $AA and increment ADD1*

*ADD1=ADD2? // check to see if we reached the end*

*If yes branch to R // done writing, now read*

*Branch to W // keep writing*

*R Reset ADD1*

*R2 CMP (ADD1)+ with $AA // check to see if the memory has what we wrote*

*If no, branch to filed*

*ADD1 = ADD2?*

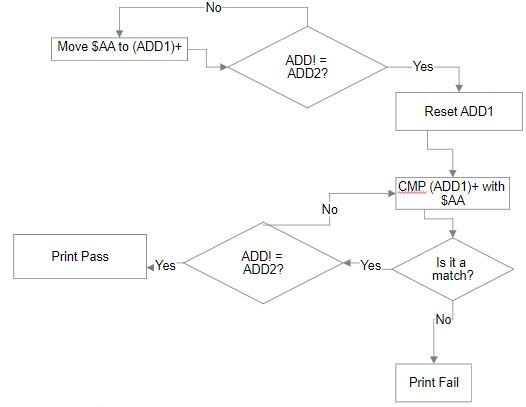
*If yes branch to pass // we have reached the end without failing*

*Branch to R2 // keep reading memory*

*Pass print passed*

*Fail print failed*

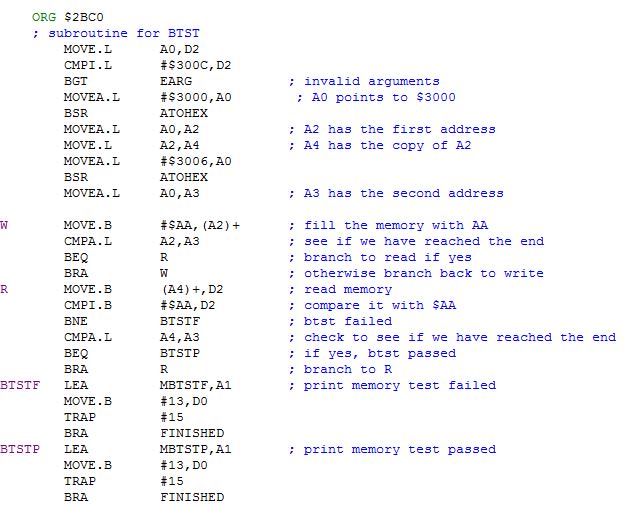
*Figure 2.26 BTST algorithm*



*Figure 2.27 BTST flowchart*

The flowchart and the algorithm explain how BTST was implemented. First, it fills the block with $AA. Then it reads the contents of the memory. It compares it with $AA. If it is not AA, it prints Failed. If it is $AA, it keeps comparing it with the next memory location until it reaches the end.

***2.2.8.2-) BTST Assembly Code***



*Figure 2.28 BTST assembly code*

***2.2.9-) BSCH***

BSCH searches a string in memory. The syntax for this command is this: BSCH $4000 $4010 hello. This will search the memory from address $4000 to $4010 for the string “hello”. If it finds it, it will print the string and the address at which the string is located. If it doesn’t find the string it will simply print, “No string found.”

***2.2.9.1-) BSCH Algorithm and Flowchart***

Before we start writing the algorithm, we will put the first argument that the user types in A2, the second argument in A3, and the string that they type in address specified by A0.

*Make D3 = 0 // make D3 a counter*

*KCMP Look at a char from ADD1*

*Increment ADD1 // find out what the first char is in ADD1*

*Compare it with first char of string // do the comparison*

*If not equal go to NOEQ // comparison failed, move on*

*If equal compare char with null // to see if we have reached the end of string*

*If they are equal branch to EQ // if yes, we have found the string*

*Increment string // to look at next char from the string*

*Increment D3 // D3 has the length*

*ADD1 = ADD2?*

*If yes, branch to FIN //reached end address but not end of string.String not found*

*BRA KCMP // keep comparing*

*NOEQ Move 0 to D3 // reset counter*

*Go back to beginning of the string // compare the string again*

*ADD1 = ADD2? // reached the end*

*If yes, branch to FIN // didn’t find the string*

*BRA KCMP // haven’t reached the end. Keep comparing*

*Yes Sub 1 from ADD1 // to get to the end of string*

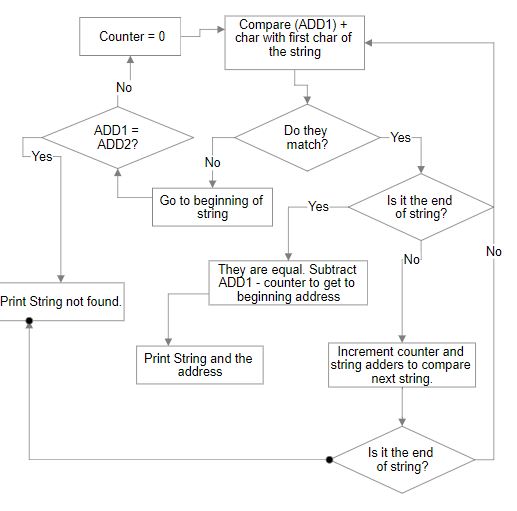
*Subtract D3 from ADD1 // to get to beginning of the string address*

*Print ADD1 // print the address*

*Print the string*

*Branch to start*

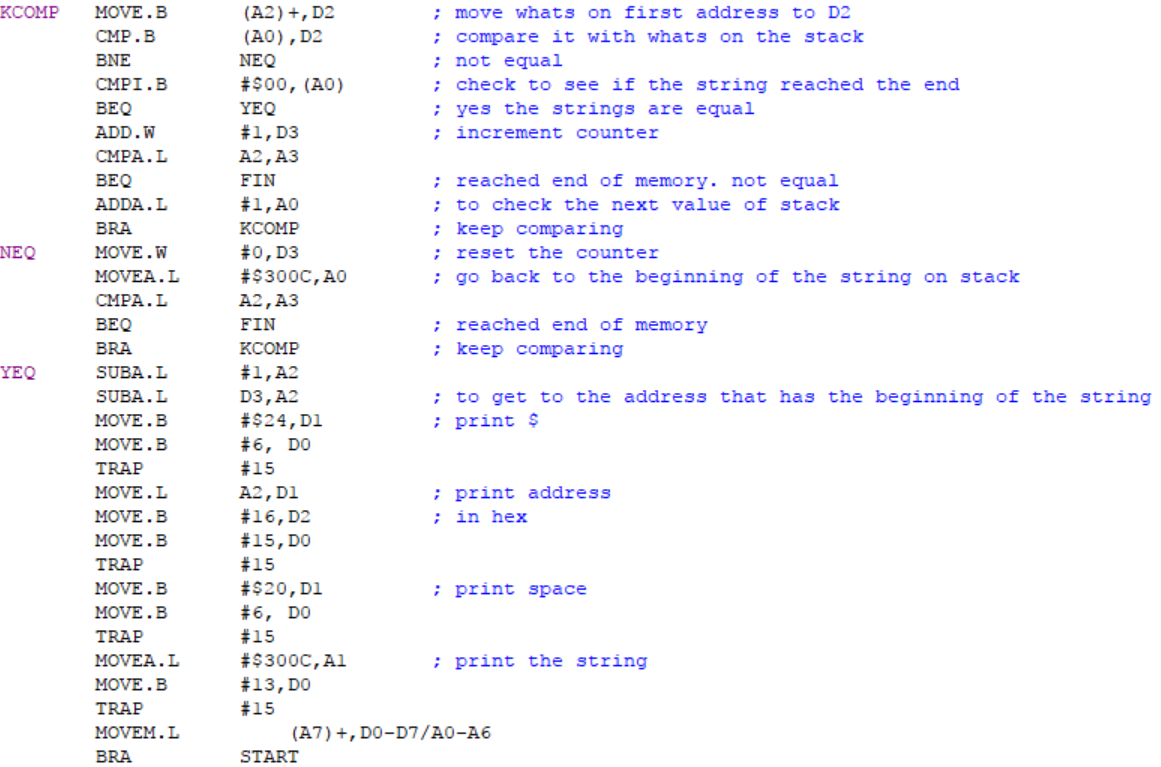
*Figure 2.29 BSCH algorithm*



*Figure 2.30 BSCH flowchart*

***2.2.9.2-) BSCH Assembly Code***

The code that is shown below implements the algorithm that is shown in figure 2.29. The part where we store the arguments in the register is not shown. A2 has the first address, A3 has the second address, and A0 has the string.



*Figure 2.31 BSCH assembly code*

***2.2.10-) Go***

The go command simply branches the address specified and executes the code that is in that address. So, if the user types in GO $4000, it will execute the code that is stored in GO.

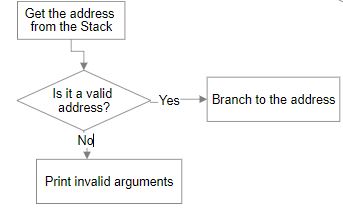
***2.2.10.1-) GO Algorithm and Flowchart***

*Store the argument in A1 // print the address*

*Convert the argument from ASCII to Hex // using ATOHEX subroutine*

*Jump to that address*

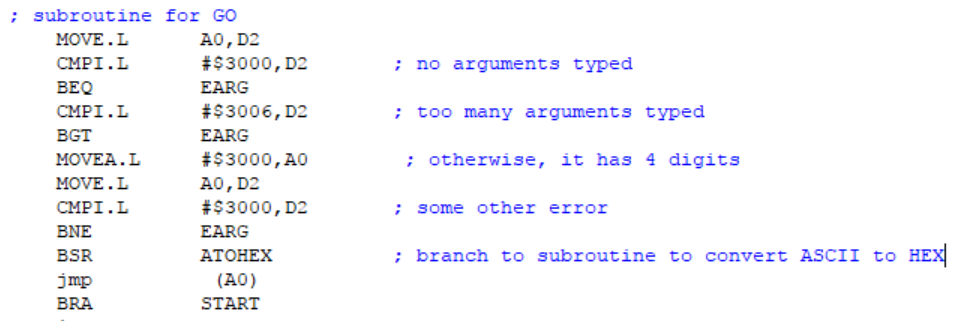
*Figure 2.32 Go algorithm*



*Figure 2.33 Go flowchart*

***2.2.10.2-) GO Assembly Code***

The code for Go is very simple. It just jumps to the address that the user passed as an argument. Most of the code is error checking for the argument and making sure the user typed in a valid address.



*Figure 2.33 Go Assembly code*

***2.2.11-) DF***

The DF command prints out the contents of all the D and A registers. It also prints out the value for PC, SR, US, and SS. This is very useful if you want to know what the status of all the registers is at a given point. This is also very helpful during exceptions because it lets us know what the values were for each register right when the exception occurred.

***2.2.11.1-) DF Algorithm and Flowchart***

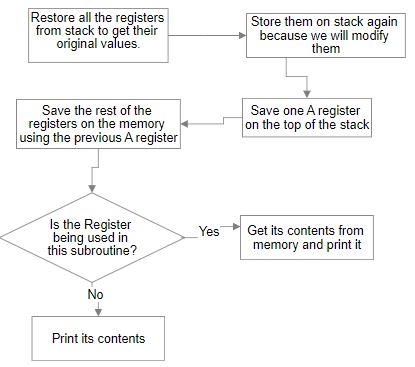
The algorithm for DF is pretty straightforward. We print the contents of all the registers one by one. If there are registers whose values are being modified during the execution of DF, then save the values of those registers in memory and print it from there.

*Is this register Being used in DF // the values of this reg will be modified.*

*If yes, save the value in memory // so save it in memory*

*Print the values of all the registers*

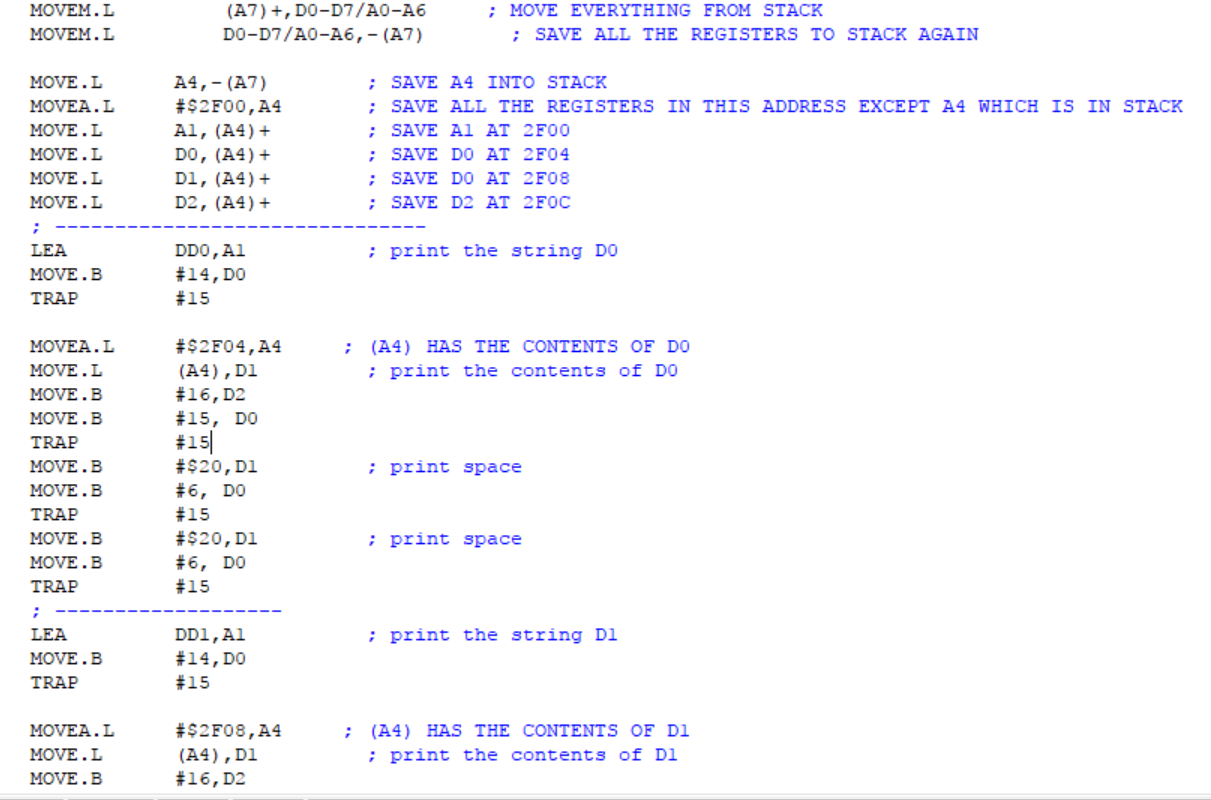
*Figure 2.34 DF algorithm*



*Figure 2.35 DF flowchart*

***2.2.11.2-) DF Assembly Code***

Our program used some of the registers and the values of the registers were modified by our program. So, we will take the original values of the registers from the stack and then we will save them on the stack again so they don’t get modified. Now that we have the original values for all the registers restored, we can start printing them. But first we need to save the values of registers that are being modified by DF. We will first store A4 in stack, and then we will store A1, D0,D1, and D2 at the address specified by A4. We will then restore this value when we are about to print them. The code that is shown below saves the registers being used to memory and then prints the first few registers. The rest of the code is very similar because we are just printing more registers using the same code. Because of its similarity to the code shown below, the rest of the code isn’t shown here.



*Figure 2.36 DF Assembly code*

***2.2.12-) Exit***

Exit command just exits the program

***2.2.12.1-) Exit Algorithm and Flowchart***

To exit the program, we just branched to the end of the code. This will exit the Monitor program.

*Jump to the label Done // no code at label Done.*

*Figure 2.37 Exit algorithm*

***2.2.12.2-) Exit Assembly Code***

BRA DONE ; branch to done

This will branch to label done, there is no code there so it will just end the program.

***2.2.13-) MOVEA***

This command moves an immediate value to one of the A registers. The immediate value must be a hex address. The syntax for this command is: MOVEA $2000 A1. This will more $2000 in address A1.

***2.2.13.1-) MOVEA Algorithm and Flowchart***

*Put the first argument in A register // save the address*

*Put the second argument in D register // where we want to save the address*

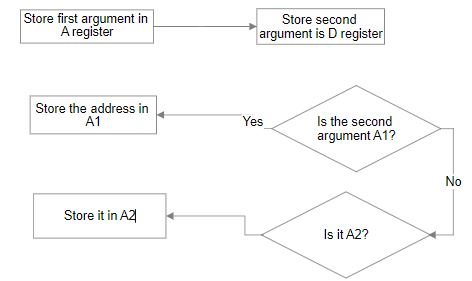
*Is the second argument “A1” // save the address in appropriate register*

*If yes, save address in A1*

*Is the second argument “A2”*

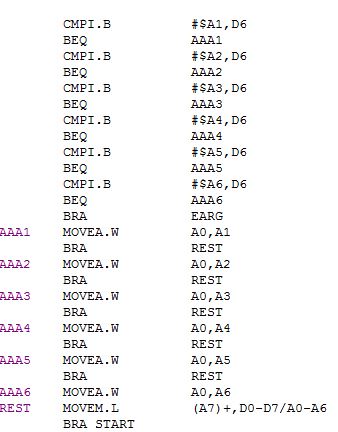
*If yes, save the address in A2*

*Figure 2.38 MOVEA algorithm*



*Figure 2.39 MOVEA flowchart*

***2.2.13.2-) MOVEA Assembly Code***



*Figure 2.40 MOVEA Assembly code*

***2.2.14-) CMPI***

This compares an immediate byte value with a D register and tells you if they are equal or not. The syntax is: CMPI $55 D5. The immediate value can’t be greater than a byte.

***2.2.13.1-) CMPI Algorithm and Flowchart***

*Restore registers from the stack // so we can look at the original values*

*Save the first argument in a D0 register*

*Compare the second argument with “D1” // compare second argument with string*

*If match, CMPI D0 with D1 // of D1.*

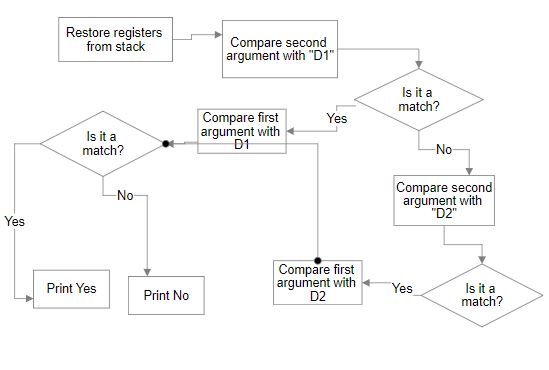
*Print yes if equal, else No*

*Compare the second argument with “D2” // compare second argument with string*

*If match, CMPI D0 with D1 // of D2*

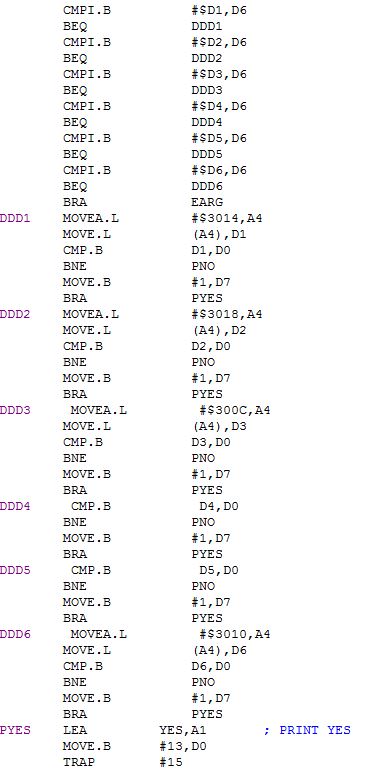
*Print yes if equal, else NO*

*Figure 2.41 CMPI algorithm*



*Figure 2.42 CMPI flowchart*

***2.2.13.2-) MOVEA Assembly Code***



*Figure 2.43 CMPI Assembly code*

***2.3-) Exception Handlers***

Exception handler routines will get executed whenever there is an exception. Whenever the exception occurs, instead of just crashing the Monitor program, the appropriate exception handler routine will get executed. This routine will tell the user what exception occurred. It will also call a DF command that will print the contents of all the registers. This will help the user with debugging. It will tell the user what caused the exception and what were the contents of the registers at that time so the user could fix the issue.

***2.3.1-) Bus Error Exception***

A bus error exception occurs whenever anything prevents the completion of a bus cycle. There could be a number of things that can cause a bus error. This includes: illegal memory access, trying to read memory at non existing address, memory privilege violation, and double bus fault.

***2.3.1.1-) Bus Error Exception Algorithm and Flowchart***

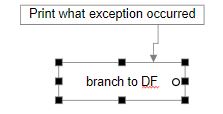
To implement buss error exception, we needed to change the exception vector table. Whenever an exception occurs, the processor looks at the exception vector table and executes the exception handler subroutine for that exception. We had to change the address for subroutine of each exception to address of our exception subroutine for that exception. And in our subroutine, we will print a message to let user know where the exception occurred and then we will call DF to print the contents of each register. This algorithm will be followed for each of the eight instructions.

*Update the exception vector table with our subroutine addresses*

*Whenever the exception subroutine gets executed call print appropriate message.*

*Call DF to print all the register contents*

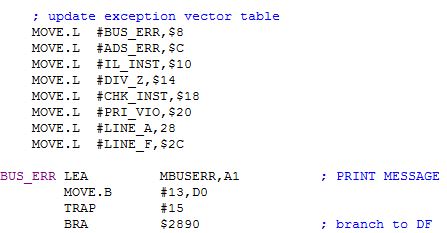
*Figure 3.1. Buss Error Exception Algorithm*



*Figure 3.2 Bus Error Exception Flowchart*

The flowchart in figure 3.2 shows what happens during an exception handler routine. This routine will only get executed once the exception vector table is updated. The code for the subroutine and updating the exception vector table is shown in figure 3.3.

***2.3.1.2-) Bus Error Exception Assembly Code***



*Figure 3.3 Exception handler subroutine assembly code*

Figure 3.3 shows the code for setting the exception vector table and for handling the buss error. The bus error handler just prints the string, “A bus error jus occurred” and then calls DF. The code for all the other exception routine will be very similar to this. Thus, I decided to omit the code for the rest of the exception handler routine to avoid redundancy.

***2.3.2-) Address Error Exception***

Address error could be cause by many things. It is mostly cause by trying to execute an instruction at an odd address. Another cause could be because if the user tried to use an illegal addressing mode. For example: MOVE.B D1,A2 will give you an address error because A register can’t be a destination for MOVE instruction. To move something in A register, you will have to use MOVEA instruction.

***2.3.3-) Illegal Instruction Exception***

Illegal instruction exception occurs when you execute an illegal 68K instruction. There are three op code that are reserved for illegal instruction error. Those three op codes are $4AFA, $4AFB, and $4AFC. Any one of these op codes will trigger an illegal instruction error. These op codes are usually generated when the op code can’t be decoded to a real MC68K instruction.

***2.3.4-) Privilege Violation Exception***

This exception occurs when you try to do something in user mode that can only be done in supervisor mode. For example, if you try to change status register in user mode, you will get this exception. This exception can’t be invoked in supervisor mode because supervisor has all the privileges.

***2.3.5-) Divide by Zero Exception***

This exception occurs when you try to divide by 0.

***2.3.6-) Line A and Line F Emulators***

Op codes that begin with A or F require a special handler. If the handler isn’t installed on the system, Line A or Line F exception will be thrown.

***3-) Discussion***

The hardest part about this project was converting the input from user from ASCII to hex. In the lab, we made a similar program that converts ASCII to hex, but in that program, we only inputted decimal digits. We didn’t input hex letters A to F. Implementing that provided additional challenges. Not only that, the argument checking was also problematic because user can type in anything and our program should be able to handle that. It makes testing really time consuming because I had to test a lot of different scenarios and type every possible combination of inputs I could. Not only this, but easy68k software also made testing really hard. It has an excellent feature of setting breakpoints that lets you stop the execution and step through your program step by step, but the thing is if you find your mistake and go back to fix it, you will have to reset all the breakpoint and do it all over again. And if the solution didn’t work and you have to modify something again, you will have to set up the breakpoints again. This was very frustrating in a long project like this because there are so many lines of code and scrolling through them to find the place to mark a breakpoint takes a long time. Other than easy68k, just the nature of assembly language made the project difficult to implement. Simple things like printing a string or having for or while loop requires a lot more thinking and programming in assembly language when compared to higher level languages. Coming up with algorithms was challenging, but to implement them in assembly, and then test it using easy68k made it a lot tougher than it had to be.

***4-) Feature Suggestions***

This project is already too long, so I can’t really suggest any idea to expand it. But I think giving the students more options in terms of selecting the commands would make it more enjoyable. So instead of having two custom commands, let students have 4 custom commands that they can implement would be better because they will get to program something they picked. Or you could add some hardware to the project because we usually worked with hardware in the lab and it was fun.

***5-) Conclusion***

This project helped me learn a lot about how the software that we use in the lab works. Designing my own version of TUTOR, which is a really good software, was a lot of fun. Even though it was frustrating at times, the end product made me feel proud of my work,

***6-) References***

[1] Motorola Programmer’s Reference Manual (Includes CPU32 Instructions)

<https://www.nxp.com/docs/en/reference-manual/M68000PRM.pdf>

[2] Easy68k Text I/O Trap Functions

http://www.easy68k.com/QuickStart/TrapTasks.htm