Specification

Specification:

1) Write an inverse assembler (disassembler) that will convert a memory image of instructions and data back to 68000 assembly language and output the disassembled code to the display. You will not be required to disassemble all of the instructions and addressing modes. The list of instructions and addressing modes is given at **Required Opcodes (https://canvas.uw.edu/courses/1408528/pages/required-opcodes)** page. Note that I'm not going to fill the memory with garbage!

2) If you want to see how a disassembler works, just take one of your homework problems and load it in memory at an address after your program. Then open a memory window and see the code in memory. You can also view it as disassembled code in the simulator.

3) DO NOT USE THE TRAP FUNCTION 60 FACILITY OF THE SIMULATOR. You must completely develop your own disassembler algorithm. If I suspect that you used TRAP function 60 I will use a search tool that I designed to scan your source code for the TRAP function 60 calls. **You can only use the simulator's text I/O function, Trap Function 15, and you can only use tasks with ID 0 to 14 of Trap Function 15. Task 15 and higher of Trap Function 15 cannot be used.**

Please check the available tasks of Trap Function 15:
http://www.easy68k.com/QuickStart/TrapTasks.htm (http://www.easy68k.com/QuickStart/TrapTasks.htm)

4) Your program should be written from the start in 68000 assembly language. Do not write it in C or C++ and then cross-compile it to 68000 code. It is really easy to tell when you've written it in C and it probably won't save you very much time. When you are working at the bit level C is just structured assembly language (or so they say).

5) Your program should be ORG'ed at $1000.

6) At startup, the program should display whatever welcome messages you want to display and then prompt the user for the **starting location and the ending location** (in hexadecimal format) of the code to be disassembled. **You need to clearly specify the expected input format for user inputs. If it's not clearly specified, then any encountered problems will get your points off.**

The program should scan the memory region and output the memory addresses of the instructions and the assembly language instructions contained in that region to the display. You should be able to actually disassemble your own program to the display! **DO NOT embed test code into your disassembler code!**

7)The display should show one screen of data at a time, hitting the **ENTER** key should display the next screen of information.

8) The address should include 8 digits completely. For example, 00001234 is an expected address format, but 1234 is not. You don't need to add 0x, $, or any other special character in front of the address.

9) The program should be able to realize when it has an illegal instruction (i.e, data), and be able to deal with it until it can find instructions again to decode. Instructions that cannot be decoded, either because they do not disassemble as op codes or because you aren't able to decode them should be displayed as:

```
00001000    DATA    $WXYZ
```

where $WXYZ is the **8-digit hexadecimal number** that couldn't be decoded. **Your program should not crash because it can't decode an instruction.** Remember, it is perfectly legal to have data and instructions interspersed, so it is very possible that you will hit data, and not an instruction.

10) Address displacements or offsets should be properly displayed as the address of the branch and display that value. **It's the absolute address value (8-digit hexadecimal format), not the displacement value.** For example:

```
00001000        BRA    000009AB        * Branch to address 000009AB
```

11) You should do a line by line disassembly, displaying the following columns:

a- Memory location b- Op-code c- Operand

12) When it completes the disassembly, the program should prompt the user to disassemble another memory image, or prompt the user to quit.