Disassembler Project Documentation
CSS422 Final Project
Team 4: Arkar Kyaw Swar, Phu Thinh Dang, Siddharth Rao

**Project Objective**

The disassembler program scans a section of memory provided by the user then decodes it into valid 68k assembly instructions.

**Project Description**

There are 3 parts that we have to build for the disassembler: input/output (IO), opcode, and effective address( EA).

IO: the program starts by asking the user to enter the starting address and ending address. There is information displayed at the beginning of the prompt to inform the user of how to enter valid input. In case, the user has entered the invalid addresses. The error will show up and ask the user to re-enter the input again.

Opcode: an instruction is identified by its opcode ( the first 4 bits in 16 bits) . However, instructions which have the same opcode are grouped together. They will be decoded after further opmode check.

EA: effective addresses are the last 12 bits in the 16 bits of the instruction. Decoding EA accurately, we know which is addressing mode of the instruction, what is the source and the destination registers.

We are using Test_demo S68 provided from the professor to test and validate our program throughout the implementation.

**Specification**:

**IO:**

After having the starting address and ending address. The program then converts them into the hexadecimal value and stores the starting address and ending address respectively in D5 and D6.

**Validate the Input**

Before storing into the data registers ( D5 and D6), the input will validated thoroughly:

- Input which is less than 8 digits.
- Input which is greater than 8 digits.
- Input number or input character are not fail to convert according to ascii
    - For number it should be in the range $ [30-39]
    - For character it should be in the range $[41-46]

- Starting address is greater than the ending address.

There will be a specific error message to notify the user. The user can either re-input or quit the program.

**Screen by Screen Display**
The dismember disassembles 20 lines of codes per time. After each display, there is a message at the end to ask the user whether they want to continue to decode or not. The user can press the enter key for continuing  decoding or simply press “N” ( or “n”) to end the program.

**Ending**
After disassembling all of the instructions  from the starting address to the ending address. There is a prompt to ask the user if they want to restart the program or not. If the user presses “Y” ( or “y”). The program restarts and again asks the user for input. If the user presses “N” ( or “n”). There is a goodbye message to be displayed and the program finishes.

**Opcode**:
For the opcode, we start by looking at the first four bits of the instruction. From there we split it into categories, narrowing what the instruction could be. From there we look at other identifying features between opcodes in that category. This could be the opmode which is usually the second three bits after the opcode (the first four bits. Using this information it is able to narrow down the instruction even further until it fully settles on a singular opcode that it can print to the console. If the instruction has other aspects like a size, it will use the knowledge it already has of how the instruction is layed out to find aspects such as the size or condition and print that out to the console as well. Any parts of the instruction don’t match any of the known versions of the instruction, we then assume that it is an invalid opcode, or data.

**EA:**
After knowing which instruction the current address holds, it branches to the specific routine of that instruction and decodes the EAs from the instruction bits which usually has 16 bits. From there, if the instruction specifies that it has immediate data or absolute addressing, the respective routines increment the address register and read the extra necessary words until it has no more words to read. Then, the program branches to the next instruction routine to start the loop again.

**Demo output of the the disassembler**

```
Wellcome! This is a dissambler program from group 4
This is some information to enter a valid input:
Address entered need to be in hexedimal and in a Longword
Starting memory addrress must be greater or less than $0x1000
Ending memory address must be in range [0x1000 - 0xFFFFF0]

Please enter the starting memory address in hex: 00009000
Please enter the ending memory address in hex: 00009038
00009000 RTS
 00009002 NOP
 00009004 RTS
 00009006 LEA (A0),A0
 00009008 LEA (A5),A0
 0000900A LEA (A7),A0
 0000900C LEA (A0),A7
 0000900E LEA (A5),A7
 00009010 LEA (A7),A7
 00009012 MOVE.B D0,D1
 00009014 MOVE.B D0,(A0)
 00009016 MOVE.B D0,(A0)+
 00009018 MOVE.B D0,-(A0)
 0000901A MOVE.B (A0),D0
 0000901C MOVE.B (A0),(A1)
 0000901E MOVE.B (A0),(A1)+
 00009020 MOVE.B (A0),-(A1)
 00009022 MOVE.B (A0)+,D0
 00009024 MOVE.B (A0)+,(A1)
 00009026 MOVE.B (A0)+,(A1)+
 Do you wish to dissamble another 20 loc? Press Enter for Yes and "N" for No:
```

```
00009028 MOVE.B (A0)+,-(A1)
 0000902A MOVE.B -(A0),D0
 0000902C MOVE.B -(A0),(A1)
 0000902E MOVE.B -(A0),(A1)+
 00009030 MOVE.B -(A0),-(A1)
 00009032 MOVE.W D0,D1
 00009034 MOVE.W D0,(A0)
 00009036 MOVE.W D0,(A0)+
 00009038 MOVE.W D0,-(A0)
 We have finish the disambler. To Restart press "R", to stop press "S"
s
Thank you for using our disassembler! Good Bye.
```

**Test Plan**

The following is our program components in order, testing strategies and standards.

User input request

- This is the very first part we programmed and we tested this by prompting the user on the screen and storing the values in a memory address and checking that address in the memory view.

Conversion of user input string to hexadecimal values

- We tested this by storing the two user input addresses into separate data registers and seeing if the converted hexadecimals match the values that the user inputted.

Loop through each memory address from starting to ending address

- We implemented a basic “while” loop which runs until the current memory address is equal to the ending address that is stored in a data register.
- Then, we tested this part by looping through the addresses and printing them all out line by line on the screen to make sure the loop works.

Read opcode

- We read the memory address with post-increment and store the opcode in a data register.
- Then, we load the L68 file of the test file and see if the stored opcode matches with the actual opcode.

Decoding the opcode and the EAs

- We started off by just testing the required opcodes and EAs one by one with a test file that only contains a few lines of code at a time. This way, we can debug more effectively and easier.
- We tested the “valid” opcodes and EAs first.
- After some time, we added the “invalid” opcode handling and tested with the undesired opcodes.

Add 20 lines per screen functionality

- We added this as the last part and just tested it with the demo_test.X68 provided on Canvas to see if it works.

Finally, with 20 lines per screen functionality, we could see how our program works with the demo_test file and fixed bugs.

Coding standards: We used the similar titles for each opcode routines and EA routines so that two people can work together. We add comments for nearly every line of code to communicate with team members.

**Exceptions**

We were struggling with identifying invalid addressing modes. When decoding the opcodes, we are easily able to identify a malformed part in the parts of the instructions that are being read and assume it is data and print “DATA” along with the instruction to the console. But doing the same for EA is hard. And moreover certain versions of instructions only except certain EA. Unfortunately we were not able to implement full detection for bad EA. There is some detection, and when it can’t decode the instruction, it will print $WXYZ in its place. If we had more time, this is most likely the place that we would focus it on.

Also, our program mixes up the MULS and DIVS instructions with AND and OR instruction. We decided not to go into it.

**Team Assignment**

We share a fair amount of workload to implement and debug the program.
This is how we split the work and collaborate:

IO: Phu Thinh Dang(65%), Arkar Kyaw Swar(35%)
OPcode: Siddharth Rao (100%)
EA: Arkar Kyaw Swar(65%), Phu Thinh Dang(35%)

**Work Schedule from week 5**

Week 5:

- Sidd implements OP code, Thinh implements EA, and Arkar implements IO
- Figure out how to take inputs from the user
- Do more research and understand the program in-depth.
- Start implementing some of the initial code.

Week 6:

- Started implementing the I/O
- Discuss how the decoding will work

Week 7:

- Validated user input ( Thinh Dang), fix bugs and continue EA( Arkar)
- Debugged and helped each other. ( all members)
- Completed I/O skeleton

Week 8-9:

- Continue working on program implementation (all members)

Week 10:

- Wrap up every deliverables ( all members)