Program Description:

Our main design philosophy was to create as much reusable and modular code as possible as to cut down on space. This also made our program much simpler and easier to deal with because multiple subroutines would all be using the same subroutine to read in the next word or long of data for example. Our other main philosophy was to comment and provide additional instructions on every line of code. We did this because we knew from the start that we were going to have multiple different people looking and working on the same code, so understanding each other was going to be key.

Our flow chart of our program is where we sectioned out the different parts of the program, which made it easy to see what parts would require more work than others, which parts depended on others, etc. Our final design slightly shifted from our initial flow chart design, but not a huge amount.

Some algorithms we are especially proud of would be READW/READL and AddWord/AddLong. These subroutines do almost exactly what you'd expect. READW and READL when called, will read in the next word or long of data, accurately incrementing the current address counter, and placing the read data in D6. AddWord and AddLong then took that value stored at D6, and appended the entire thing to the good buffer. These subroutines are modular, and used all over the program. They are especially useful in instructions like MOVE and it's variants, because for example, when we know that the following destination or source is an absolute or immediate data of either word or long, we just call READW or READL, and then AddWord or AddLong immediately after.

The program has a very linear flow. It starts with a Jump table that looks at bits 15-12 and then it branches to the correct sub section. Once there it then adds the correct letters to the stack to be printed, or it checks other bits to determine which of the possible subcategories it is. If none of them apply, then the program branches to "INVALID" to print out the hex code.

Once we have determined bits 15-12 we use a subroutine called "SPLIT" to split up the various bits into different Data Registers.

Bits 2, 1, 0 are placed into Data Register 5.
Bits 5, 4, 3 are placed into Data Register 4.
Bits 8, 7, 6 are placed into Data Register 3.
Bits 11, 10, 9 are placed into Data Register 2.

We then compare different numbers to the data registers to see if we should print out a Byte, Word, or a Long. If there is no match, we then call the "INVALID" subroutine to print out the Hex code.

Some limitations we are aware of include the following:

- User is limited to inputting only hexadecimal words for starting and ending memory addresses
- Disassembler can only decode the instructions provided in this projects list of required opcodes and addressing modes
- All values part of an instruction decoding will be shown in hex, even if the initial instruction gave the value in decimal or binary. For example: TRAP #15 will be decoded as TRAP #$F

Specifications:

Our program disassembles binary computer code into human readable 68000 Assembly. It can translate 25 op-codes and 8 ea modes. The program prompts the user to enter a word of hex representing a starting and ending address of a section of memory. This section of memory is what the program will decode from machine code back into assembly.

As required our program starts at address $1000 and does not use Trap functions above 14 and Trap Tasks above 15. At start up the program asks the user to enter a hexadecimal address in all caps for a starting memory address and then an ending memory address. The program will then run through the memory range selected.

If the user enters a non-valid hexadecimal word like $XTXT, the program will restart, prompting the user to try again.

Thirty instructions will be printed at a time to the console. The user will need to press ENTER to tell the disassembler to advance to the next screen, showing the next thirty instructions.

If the disassembler encounters any machine code that does not represent a valid instruction the program does not crash, rather, the memory location of that machine code, the word “DATA”, and then the machine code itself will be printed to the console.

If the disassembler successfully decodes an instruction, the memory location of the instruction will be printed, followed by the decoded instruction. If needed the program will place the ea and data registers in the correct position. For example if SUB is printed it will correctly show Dn - <ea> → Dn *or* <ea> -Dn → <ea> depending on what the binary code shows.

The program also prints the instructions to the console window with the same spacing that a programmer would use for in a 68k text editor. The program only prints off the

code and not the comments because they are removed by the compiler prior when the program runs.

Once the disassembler has reached the ending address provided by the user, the user will be asked if they want to disassemble another section of memory. If the user chooses yes, the whole process repeats. If the user chooses no, the program gracefully quits.

Test Plan:

We created a testing file that contained every form and permutation of every instruction we were required to decode. We called this file testing.X68. During the final phases of our instruction, we loaded and tested our program with this file multiple times, making sure that every instruction was either being successfully decoded or displayed as DATA if invalid.

From the provided specification for the final project we were limited to specific op-codes and effective addressing modes. The 25 instructions we needed to decode were:

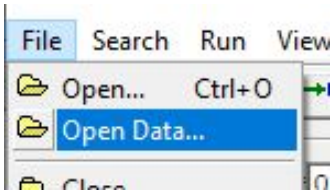
- NOP
- MOVE, MOVEQ, MOVEM, MOVEA
- ADD, ADDA, ADDQ
- SUB
- LEA
- AND, OR, NOT
- LSL, LSR, ASL, ASR
- ROL,ROR
- BGT, BLE, BEQ
- JSR, RTS
- BRA

For each instruction with multiple available effective addressing modes, we were required to implement 8 addressing modes:

- Data Register Direct
- Address Register Direct
- Address Register Indirect
- Immediate Addressing
- Address Register Indirect with Post-incrementing
- Address Register Indirect with Pre-decrementing
- Absolute Long Address
- Absolute Word Address

These requirements on what we needed to be able to disassemble gave us a clear picture of how to create a testing file. Our testing file contains at least one case for every valid addressing mode and instruction combination along with edge case testing for maximum register numbers and memory ranges that could be included in the instructions.

With our testing file instructions ready, our testing file is organized at a memory address outside the range of memory addresses consumed by the disassembler itself. In particular, we chose to load the testing data at address $3000 with “ORG $3000.”



With any new iteration of our testing file, we generated a .S68 that we loaded into memory for use during an execution of the disassembler. Our method of loading the .S68 data into memory was with File > Open Data… in SIM68K. Once the data is loaded in memory we are able to specify it in our memory address range input and then visually inspect that the output of the program matches our assumption using the test file .L68 as a reference.

Exception Report:

The biggest hurdle we had to face was the lack of good and specific information about how to code in 68k.

During some of our debugging sometimes the ez68k program would auto optimize code and turn ADD into ADDI which makes the program faster, but also means that our disassembler cannot convert it, so it shows up as “DATA” and a hex code.

There is technically a way to make the program interpret a range of memory as containing instructions it does not have. This happens through instructions not included in the set of instructions from the specification. When an instruction is outside the specification and is over one word in length, further data following the opcode word that represents the same set of bits as an instruction we would normally disassemble. We’ve seen this case happen for MOVE.B and MOVEM.

Team Assignments and Report:

Christopher and Paul were on the OPCode team, and Mitchell and Matthew were on the EA team.

We all collectively worked on the I/O skeleton before splitting up into teams.

1) Christopher - 25%

- Created and filled out jump table and CODEXXXX subroutines (opo-code portion). Filled out the initial decodings of instructions, then EA team built on top of them.

2) Paul - 25%
    - Created I/O input validation, program looping, some op-code reads including BCC and shifts/rotates, paging of decoding process (30 instructions per page advanced by user hitting enter).
3) Mitchell - 25%
    - Responsible for EA decoding of **NOP, MOVE,MOVEQ,MOVEA,MOVEQ, LEA,JSR,RTS, BRA,BEQ,BGT,BLE, AND,OR,NOT** instructions.

4) Matthew - 25%
    - Responsible for EA decoding of **ADD,ADDA,ADDQ, SUB, LSL,LSR,ASL,ASR,ROL,ROR** instructions.