Group B10: Documentation for Disassembler Project
Yeseul(Ashley) An, Chandler Ford, and Catherine Santos

1) **Program Description**

Design Philosophy
Our overall design philosophy for this project consisted of two main components: modular programming and procedural programming.

First and foremost, we focused a lot of our time and effort on making our program modular. This influence can be observed in the names of our files. They each have a main task or purpose that is reflected in whatever they are titled, as explained by the examples below:

- Branches.X68: This file concerns itself solely with finding out the correct opcode so the program can branch to that specific opcode label.
- Definitions.X68: This file handles definitions and defining the messages that get printed to the console.
- EAs.X68: The file decodes effective addresses.
- Op-code.X68: This file disassembles and prints opcodes.
- B10_Testfile.X68: The file tests the disassembler program.

We also adhered to procedural programming practices by incorporating subroutines to handle repetitive tasks, as opposed to hardcoding everything manually. Our file called Subroutines.X68 handled all sorts of our program's conversion, evaluation, and printing functionality. For example, we coded a subroutine to handle moving the information stored in bits 11 through 9 to register D3, a task that the program often had to perform.

Flowchart
We created a detailed and comprehensive flowchart for this project. We broke the process down into three main sections, which are identical to the recommended roles: input/output, effective addressing, and opcodes.

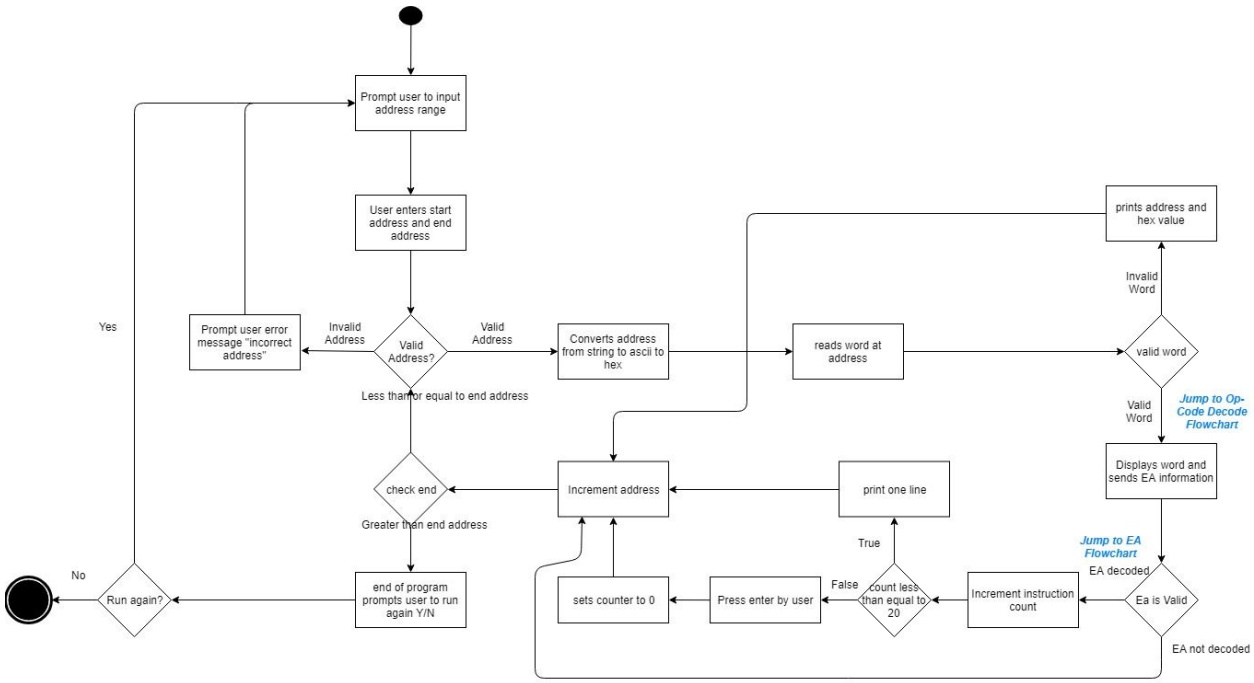

Figure 1: I/O Section of Flowchart

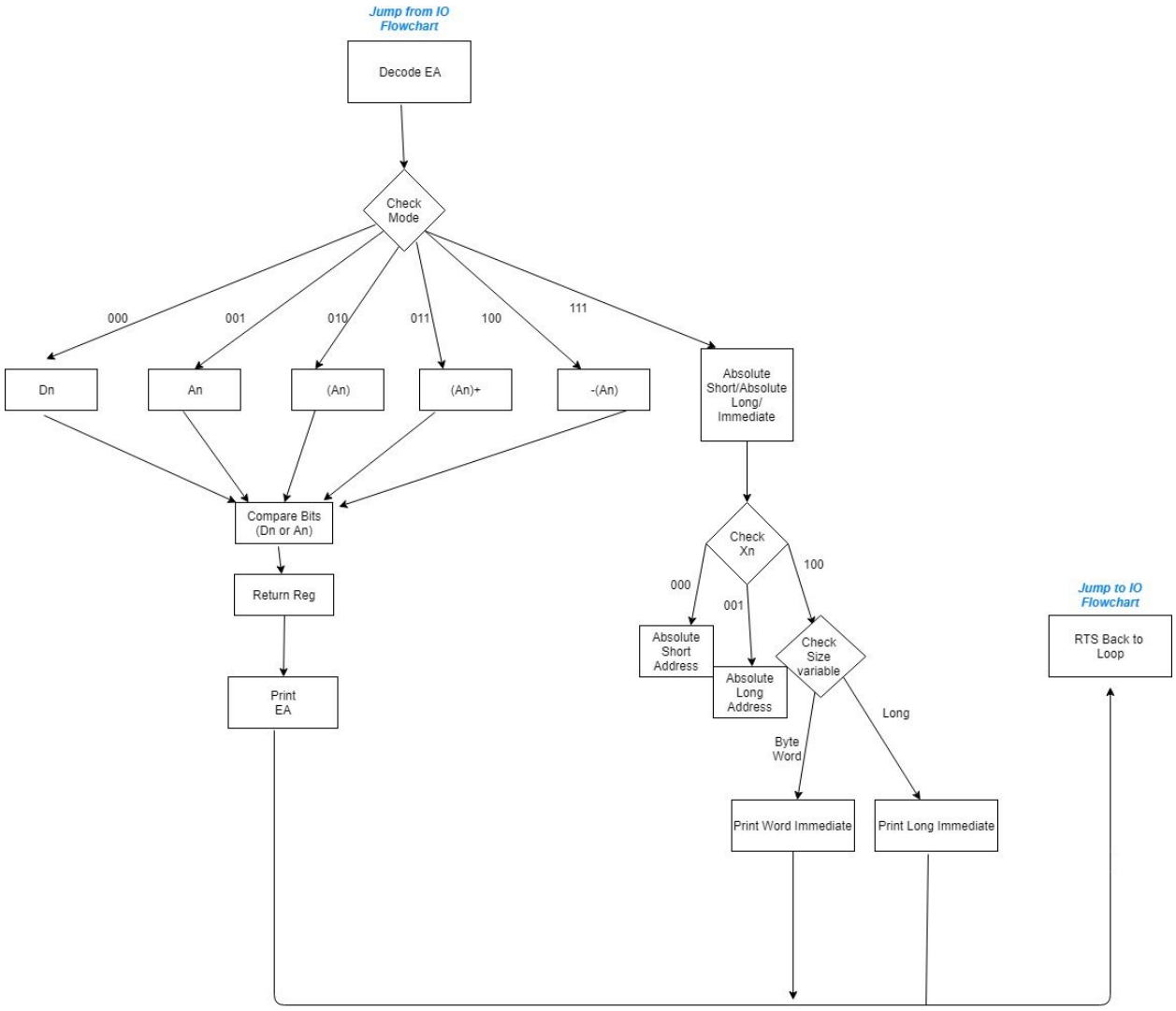

Figure 2: EA Section of Flowchart

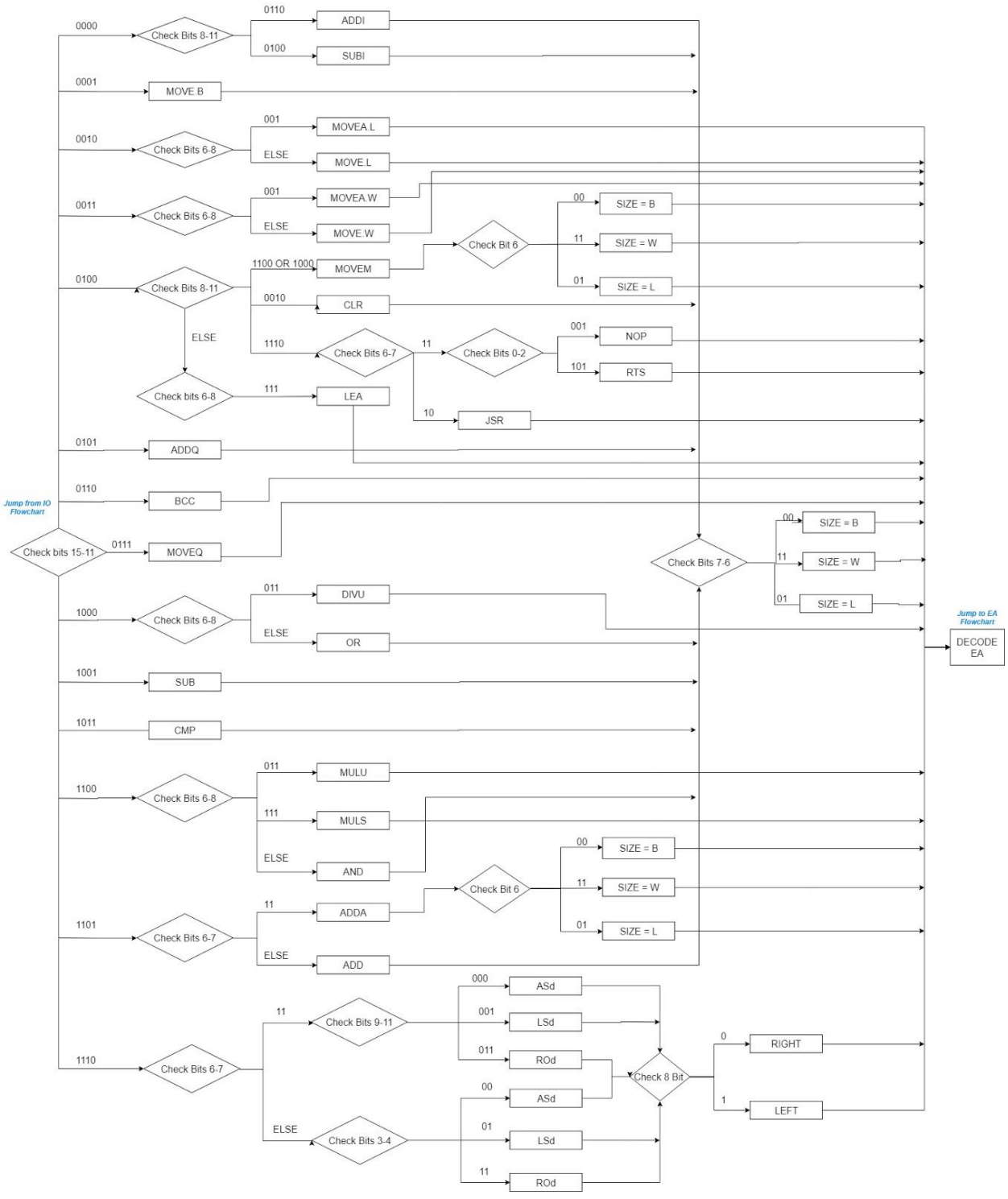

Figure 3:Opcode Section of Flowchart

To briefly summarize our flowchart, it begins in the input/output by prompting the user and waiting for input. If the address is valid, the address is converted from a string to ASCII to hex before reading the word value. Valid words have their opcodes decoded by checking different individual and/or groups of bits. The effective address process is determined by the mode. The information is printed, the display counter is modified, and the address is incremented. This process loops until the end address is reached.

Special Algorithms

We are especially proud of our algorithms that checked different groups of bits. These were located in Subroutines.X86, and were extremely helpful for decoding opcodes. For example, the 11-9 algorithm checked the bits located at positions 11, 10, and 9. The bits located in these positions often designated the register, and having a premade algorithm that could be called made the process much more efficient and simple than hard coding it by hand.

We are also proud of our algorithm, called Evaluate_Size, that evaluates the size of an opcode. It compares the value in D3 to the hex values zero, one, and two. These values correspond to byte, word, and long sizes, and the program branches accordingly. This algorithm, found in Subroutines.X86, was helpful when coding the file that dealt with opcode decoding.

For MOVEM it could be very complex to design and code the algorithm. However, we modularized the algorithm and wrote the subroutines that do evaluation of bits for different modes of MOVEM simpler and easier. This ultimately made opcode MOVEM to be printed much easier and not confused by calling just the subroutine that evaluates data register or address register for different modes of MOVEM.

Algorithms from Other Sources

When it comes to removing slashes before the comma or at the line of the MOVEM, we got an idea from the team B9, which was basically to backspace the slash and replace the slash with the new comma using EQU $08. This makes the correct output to be printed to the console.

Limitations

One of the limitations that we figured out was the source control of our codes. We used GitHub private repository because it is very well known version control service. For initial phase of our coding, it was convenient to use GitHub because it merged our codes together using short command lines and not losing any line of codes. However, in the later phase of our coding when we started to write

more complex and quite a lot lines of code, we found that GitHub's automatic merging caused some problems and costed us time to fix the problems. We decided not to use the GitHub and used Google Drive to do the source control for this project.

2) **Specification**

Our program disassembles the required opcodes by reading them as data file and store in the memory and output the opcodes to the console. Unrequired opcodes are to be read as invalid data and will be printed out as 4 hexadecimal values with its address. Our program recommended the users to start at the address $7000 and type 4 hexadecimals for the address. The program checks the invalid characters such as ^,&,# and asks again the address when it reads the invalid character. Also, it checks whether starting address is greater than the ending address and return to ask the address again if that is the case.

Our program has the following functionality:

Input/Output

- Prompt user to input data
- Allow user to input data for starting address
- Allow user to input data for ending address
- Convert address from string to ASCII to hex
- Read word at address
- Display word and send effective address information
- Print up to 20 lines per section
- Prompt the user to press enter to continue printing
- Increments address until end of program reached

Opcodes

- Support for all 28 required opcodes
    - NOP
    - MOVE
    - MOVEA
    - MOVEQ
    - MOVEM
    - ADD
    - ADDA
    - ADDI
    - ADDQ

    - SUB
    - SUBI
    - MULS
    - MULU
    - DIVU
    - LEA
    - CLR
    - AND
    - OR
    - LSL
    - LSR
    - ASR
    - ASL
    - ROL
    - ROR
    - CMP
    - Bcc (BCC, BGT, BLE)
    - JSR
    - RTS
- Ability to branch program to the correct opcode
- Ability to decode the size of the opcode
- Ability to store the register/count
- Ability to store the mode
- Ability to print opcodes, sizes, and tabs
- Ability to send information to the EAs file to be decoded

Effective Addresses

- Support for all eight required effective addressing modes
    - Data Register Direct
    - Address Register Direct
    - Address Register Indirect
    - Immediate Data
    - Address Register Indirect with Post incrementing
    - Address Register Indirect with Pre decrementing
    - Absolute Long Address
    - Absolute Word Address
- Ability to check the mode and branch accordingly
- Ability to compare bits for destination/address registers
- Ability to return register after comparison

- Ability to print effective address
- Ability to handle and print immediates
- Ability to return to main program loop

3) **Test Plan**

Description of Testing
We primarily tested the program with the demo_test.X68 file, either in its original form or as the basis for our own specialized testing files. In one such file, named B10_Testfile, we added additional lines and different variants of effective address fields to make testing more rigorous and comprehensive. Additionally, we created a file to test invalid cases to make sure the program handled the input appropriately by responding with an error.

We followed regression testing by whenever we completed one more op-code, we tested the entire opcodes that we had been developed together with the updated test file. One of the reasons was that since we are using constrained number of data registers and address registers for easy68K, sometimes the result of previous op-code still remain in the registers and affect to the disassembly of other opcodes.

For the actual process of testing, what we generally did was open up and run the main Disassembler.X68 file and add in the demo_test.S68, B10_Testfile.S68, or whatever other testing file we were currently using. After entering 7000 for the starting address and 8000 for the ending address, we ran through the program until we got the section of the output we wanted to analyze. We would then compare it to the actual file to see if it printed correctly or incorrectly.

Coding Standards
We have comments indicating the purpose of critical and/or confusing sections of code. We attempted to break larger algorithms down into smaller sections with labels that could be jumped or branched to. Each file has a header with the title, our names, and the purpose. We used a similar overall style for general formatting and indentation, as all parts of the code were tabbed appropriately. We each tried to name subroutines and labels in descriptive and consistent ways. For example, we prefaced each of the labels for our opcode printing definitions with “PRINT_” for clarity and consistency.

Include Test Files
We included B10_Testfile and demo_test in our submission.

4) **Exception report**

Unfixed Problems and Other Deviations
The main unfixed problem in our project is that the program can't handle uppercase yes or no input when the user is asked if they want to restart. However, we do prompt them that they should enter a lowercase yes or no. As far as deviations from are concerned, the program is designed to start from 7000. The program works fine when the starting address is inputted as an address before 7000 provided there is actual data there.

The program is designed to end when it reaches FFFF. There is no data before 7000, so the program assumes that the end has been reached even though that might not be the case.

What Was Completed in Time
We completed everything the assignment asked for in the time allotted.

Defects
Our program has no known defects based on our testing with our own testing file and the test_demo file, provided the starting address entered isn't before 7000.

5) **Team assignments and report**

Overall, we did our best to finish the project on time. We planned and started the project early so that we can allocate more time in testing our program thoroughly. Initially, we did not assign each person with different roles such as I/O, EA and Op-code. Instead, we started by assigning simple op-codes by each person and worked on it; this helped all of us to have a good understanding in what we are doing each other and agree on which parameters we are using for opcodes and passing to the EA. After we all had a good understanding of how the program worked, Ashley focused more on the output and EA while Chander and Catherine did opcodes. The team all worked hard to make the best by doing what we are good at. "We agree that each team member deserves an overall work percentage score of between 30 and 40% depending on the person."

Individual roles:

Yeseul(Ashley) An handled I/O and EA and some of the assigned op-codes. Completed I/O and EA while group members all were working on the assigned op-codes so that group members could see their progress in the console and made opcodes to run consecutively. Ashley handled the paging for the output window to print 20 lines with pauses in between. She fixed the major bugs when

the project was tested by the test_demo and team's test file. Ashley recorded the Log of the project to show the evidence that the program runs completely without the bugs when tested by test_demo file. She also helped fix the rotation and shifting opcodes.

Assigned opcodes: MOVEM, SUBI, ADDI, and MOVEQ, as well as handling invalid opcodes.

Percentage of Coding: 40%

Chandler Ford worked on the majority of the documentation and presentation, and also helped by contributing to the progress reports. He worked on the opcode decoding for a lot of the opcodes, including the rotating and shifting ones such as LSL, LSR, ASL, ASR, ROL, and ROR (which he also handled the branching for). Chandler added some of the bit evaluation subroutines, as well as some of the definitions for printing. He helped test the program with the different test files and found bugs to be fixed.

Assigned opcodes: DIVU, OR, SUB, CMP, MULU, MULS, AND, ADD, ADDA, LSL, LSR, ASL, ASR, ROL, and ROR

Percentage of Coding: 30%

Catherine Santos came up with the overall project design and detailed flow chart so that we can easily implement the code with very structured and modularized way. Catherine put a lot of work into making detailed progress reports. Also, Catherine created and formatted the opcode table, which are visited by our group members throughout the entire project development. Catherine implemented the branch file and JMP table for branching to decoding subroutines.

Assigned opcodes: MOVEA, MOVE, CRL, NOP, RTS, JSR, LEA, ADDQ. BCC, BGT, and BLE

Percentage of Coding: 30%