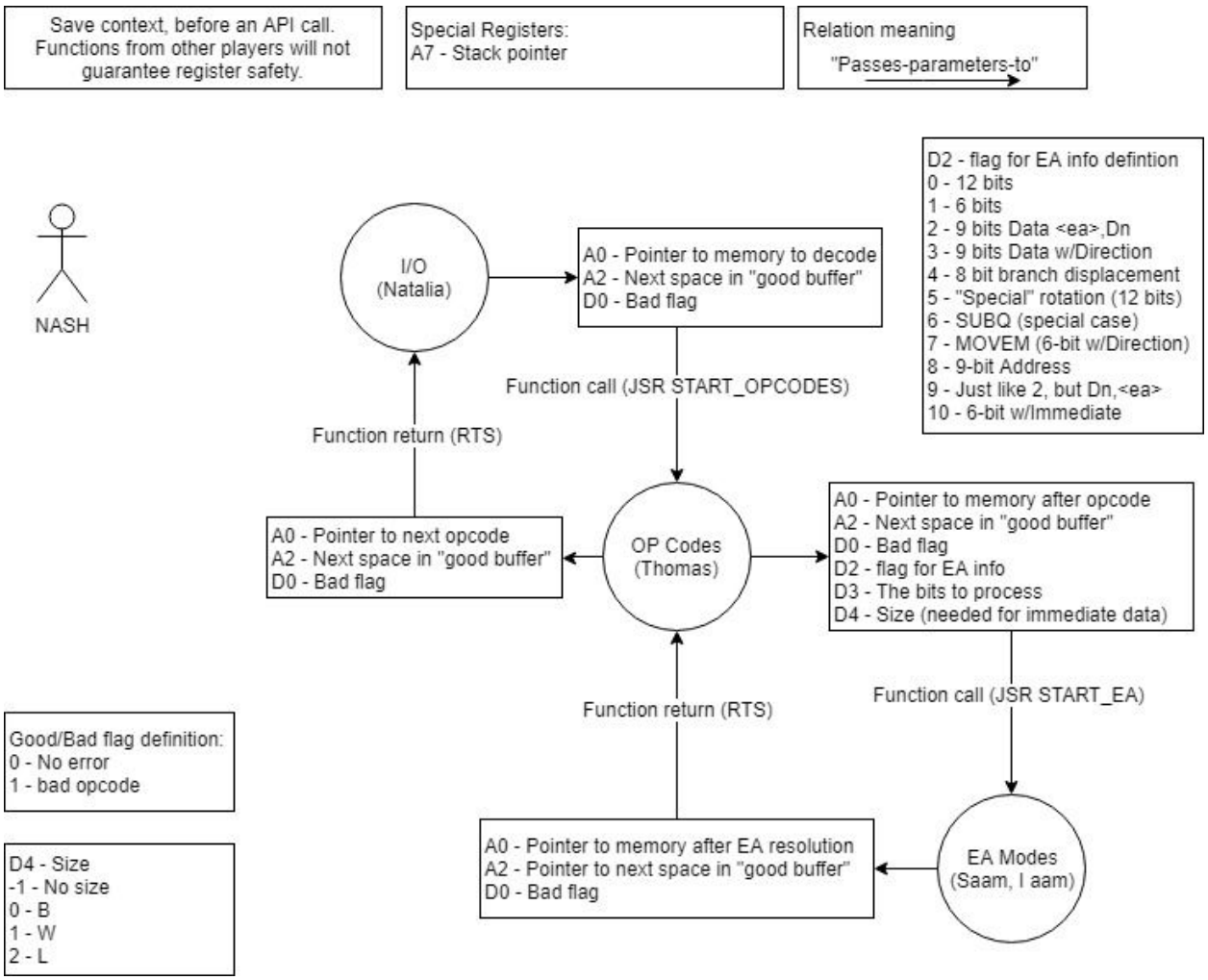
Natalia Gilbertson, Thomas Kercheval, Saam Amiri
3/16/2018
Disassembler: Project Description

We approached the disassembler by dividing it into three modules: the I/O module, the Opcode Disassembly module, and the Effective-Addressing Disassembly module. We started by defining an API so that each module knew how it was talking to the other modules:

Disassembler API



This API shows some specifics about how the modules send data between each other, e.g. which register holds the pointer to the next instruction to decode, which register holds the pointer to the output buffer. This is how we defined passing parameters to each other (when I say “we” I mean each of us took one of the modules to implement).

Now, let’s jump into the implementation of each module, starting with I/O.

I/O Module

This is the module that drives the disassembler. When I/O is run, it prompts the user to load their data-to-be-disassembled into memory, and asks the user for the starting and ending address (inclusive) of the data in memory. It checks that the addresses are valid (on a word boundary, valid address in memory) and prompts the user to re-enter starting and ending addresses with an error message if the input is invalid.

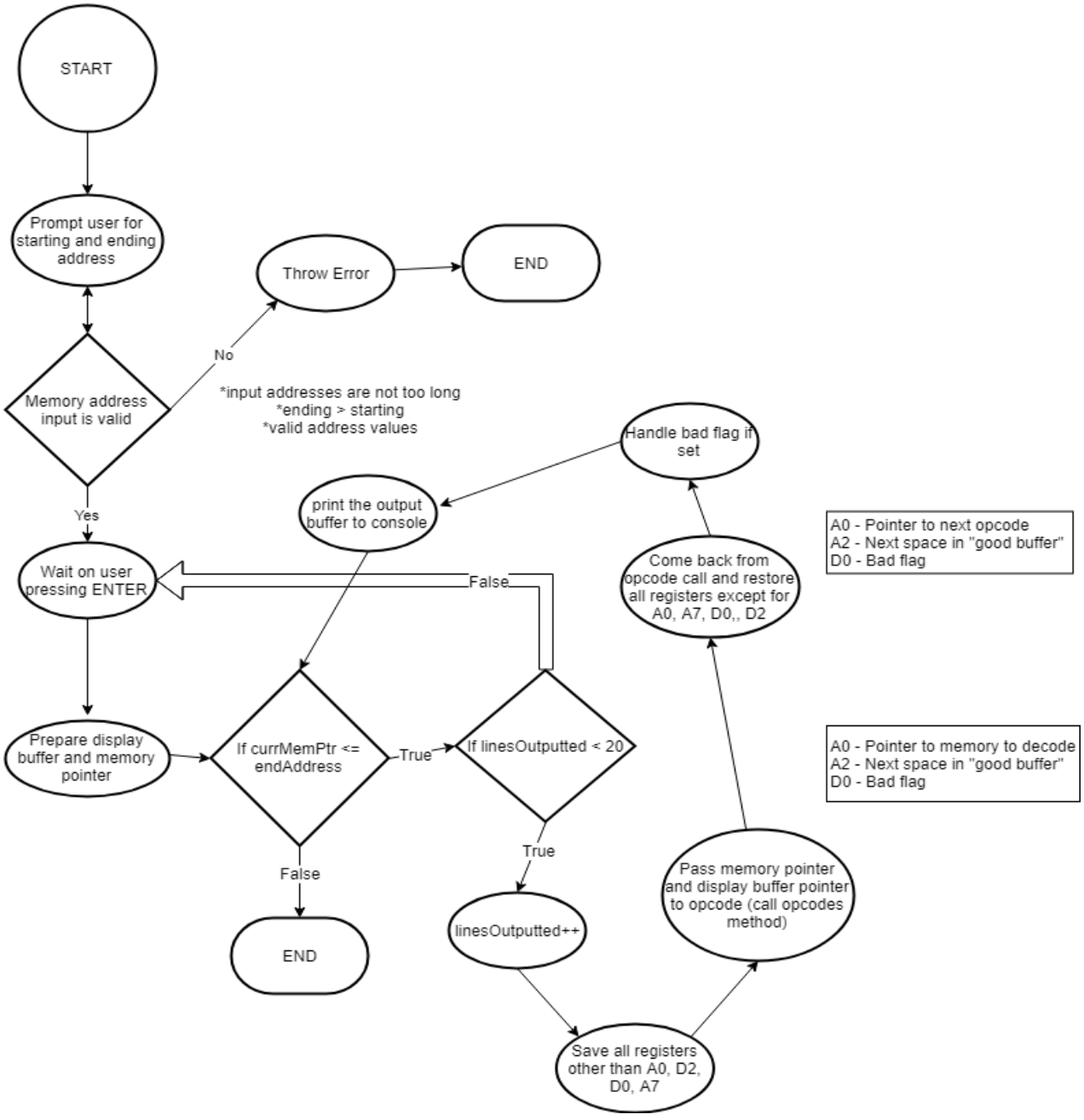
After the input is validated, the program will translate the user input, which is loaded in as ASCII, into hex values in an address register. Next, the program will prompt the user to press ENTER to print out a page of disassembly instructions. The program hangs on reading in keyboard input so that when the user presses ENTER it processes 20 instructions, printing 20 lines to the console. This means the I/O module calls JSR OP_START 20 times. If there is not 20 instructions left to disassemble, then the I/O module will know to end the program even though less than 20 lines were printed out. The program will print a nice “end of file” message when it has disassembled all the data in memory between starting address and ending address (given by the user).

Before I/O calls the opcode module, there are a few steps it has to take before making the function call. First, it needs to reset the output buffer, which we have overwriting the same place in memory for each call that processes instructions in memory. Each module in the disassembler appends characters to the output buffer based on what they disassemble.

Next, I/O puts the current address in memory, where the next opcode is stored, into the output buffer. This is a particularly clever part of the I/O module. It is implemented in the function “AddCurrAddressToBuffer”. What the function does is first it puts the address, which starts in an address register, into a data register (this is necessary to perform certain bit manipulations to the address). For all 8 hex characters of the address, the function performs the following steps: rotate the data register holding the address left four bits (one hex character), grab the first byte out of that data register and move it to another data register, mask out the second nibble of the byte (the most significant hex value in that byte), then get the ASCII value of the remaining nibble from the string hashtable and put it in the next space in the output buffer. Using the ROL.L operation the function can start by grabbing the most significant nibble/hex value of the address, processing the address for output from left to right, just how it should be printed out.

After I/O does this, it performs a caller-saved MOVEM on the registers it uses, then it calls the opcode module, returns, gets the registers back from the stack, and then outputs the buffer with the new disassembled line of code. That's the I/O module.

I/O Flow Chart



Opcode Disassembly Module

The Opcode Module detects which opcode is currently being disassembled, the size of the opcode, and which Effective Addressing (EA) format is present (passed as a flag to the EA Module). Validity is tested, so no ops that are not supported by our disassembler are incorrectly flagged as valid.

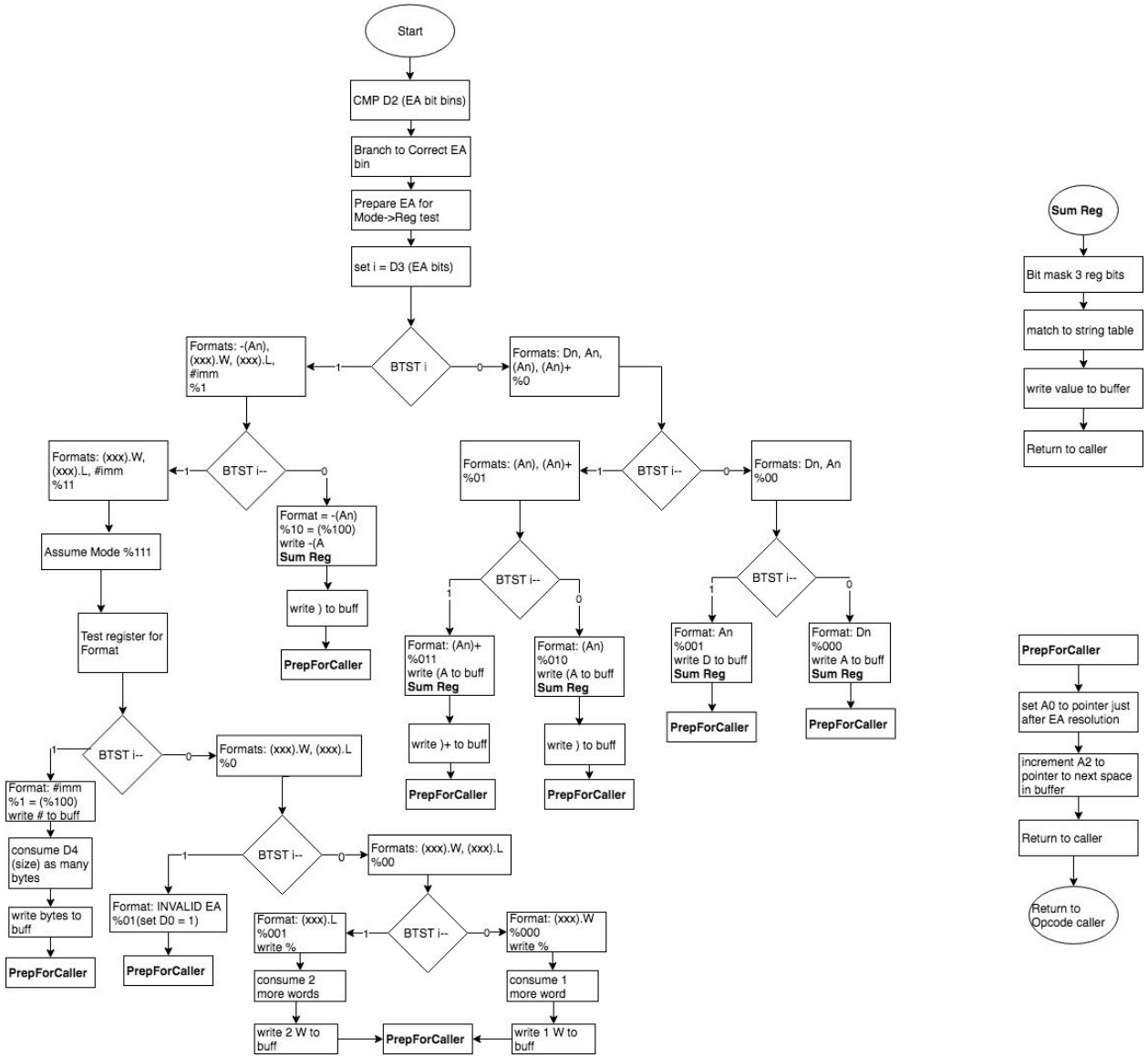
There are two “cool” pieces of this module to highlight. The 30 opcodes which are required are selected using binary search. This was by design (see the **`Opcode_Decision_Tree.png`** flowchart, in `68K_Disassembler/documentation/op_docs`, to see how this works). The binary search was done by sorting the opcodes by bit value in a speadsheet (see the **`Sorted_Opcode_Spreadsheet.pdf`** design document in `68K_Disassembler/documentation`), then designing a binary tree based on the sorting. The extra opcodes were added in an ad hoc fashion, thus are not as cool.

The other cool thing is how branch opcodes were implemented. Because the condition code is a 4-bit value and everything else is the same, a string table is used to select which branch statement is present given the binary representation of an opcode.

This module, along with all of the others, were designed before implementation. See the documentation folder in the project for more of our design documents!

(SEE NEXT PAGE FOR EA)

Effective-Addressing Disassembly Module



In my portion of the project I stressed planning over brute forcing the coding. This was done by creating an overall flow chart of the the decision that must be done based off of the data that is given to me by Thomas's module. I used the flow chart above as a 1 to 1 mapping of the code I would implement and every case that I would fall into after the branching was done. This allowed me to stay organized plan ahead of implementation. I am most proud of two portions of my code, First, the structure of my bin testing where it was self explanatory and let me know what was going on at all times of processing. Second, my implementation of MOVEM which involved its own flow chart diagram and multiple functions to implement. MOVEM felt like a program in its own as it went against the grain of all other functions I had made previously.

Specifically, I am excited to have come up with a way to reuse the core logic from MOVEM, by processing the register bit mask for both post increment and pre decrement with the same logic. This was accomplished by formatting the bit mask to match the format of the default Post increment register bit mask. The eventual goal of my module was to determine the EA modes for the instruction passed to my module and write it to the buffer with correct formatting. I feel like my module accomplishes this and goes above the expectations as it works for all specified cases as well as pointing out, out of scope EA modes that we were not assigned. This is important as I had to actually see what mode the bits broke down to. This includes out of scope modes to decide whether I would return it with the bad flag set (demonstrating an invalid instruction), or going on with processing of the EA modes format if it were valid. In short even though I did not have to do the extra EA modes I still located them and was able to tell the other modules of their existence.