Team Fireball: Binh Nguyen, Derek Willms, Brian Quigley

Program Description

Our program begins by greeting the user with a splash image and then prompting them for a starting and ending address. This address is immediately converted into ASCII. As it is being converted, it is checked whether or not it is even or odd. Odd numbers are unacceptable, so the program will error and prompt the user again. Additionally, the address must not contain lowercase hex, or have an address that is not within the specified range, or with an ending address that is equal to or less than the starting address.

Once the addresses are verified as proper, the starting and ending addresses are saved, so we can constantly evaluate whether the program has finished going through the test code. The starting address is incremented as a placeholder for the current position of the program.

Next, our program grabs the first word and decodes it. At this juncture, we check to see if the word is an instruction that our program is required to handle. If it is, then we check to see if additional words are required and grab them if necessary. If the word is not a required instruction, then we simply print “DATA” and the address it is in. Once the decoding is completed, we convert the results back into ASCII and print them in a formatted fashion. At this point, we check to see if the starting address pointer has advanced beyond the end of the test code, and make a decision whether or not to continue.

The flow chart in Figure 1 illustrates the high level events going on throughout our program.

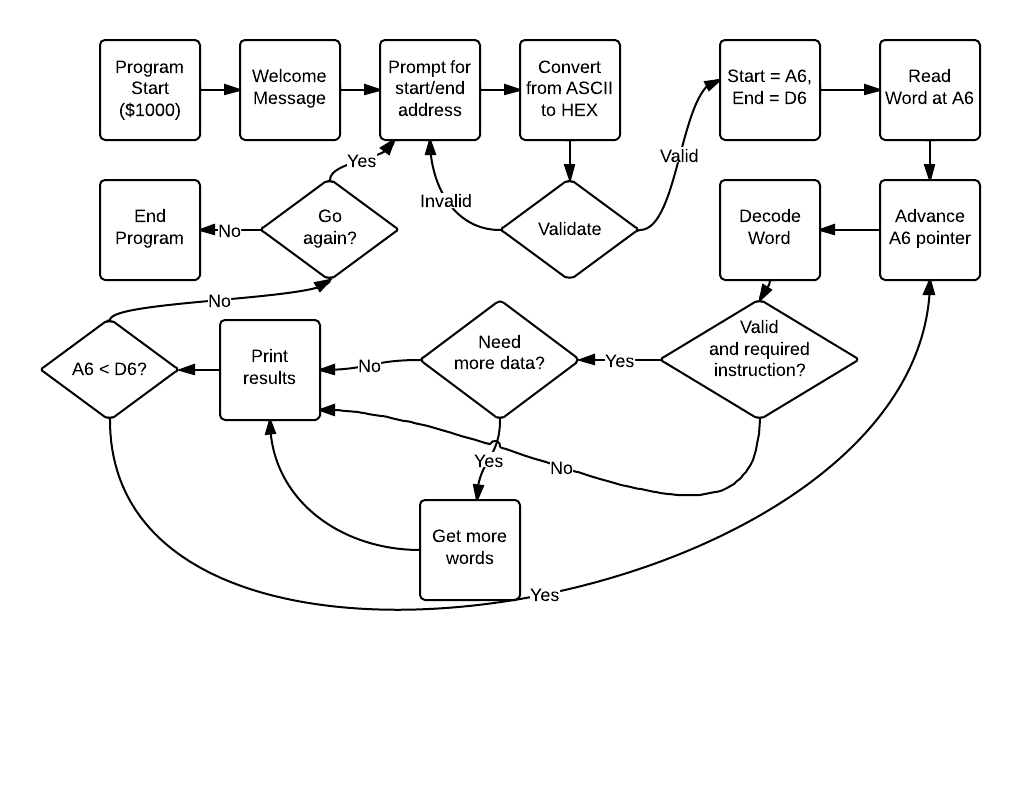


Figure 1: Flowchart

The only routines that we used that could be described as “canned” include the conversions to and from ASCII. We researched specific values that could be added or subtracted to numbers to convert them from one language to another. This also helped verify whether characters were valid. Most of our knowledge came from the textbooks.

Binh was proud of his “get\_space” algorithm that he used to format the output perfectly, because a simple tab would not have equal spacing between different outputs. It adjusts based on the max\_space variable, which dictates exactly how much room you want between each column.

Our design philosophy was to divide the group into IO, EA, and Op Code roles. However, as we got deeper into the project, it became less clear what the differences were between Op Code and EA roles. We spent a lot of time agreeing on what variables were best suited for each value, and tried to communicate what variables our subroutines required. Our first goal was to create a skeleton program that could take in a test program and decode a simple MOVE command. From there, we also did ADD commands and proceeded to fill in the rest of the instructions.