# Program Description

## Goals

Before we started programming our project, we established some goals for its development:

### *Redundancy Goals*

Keeping redundancy to a minimum was a main goal. We set out to do this by creating as many utility subroutines as possible so that our overall code would be light and flexible.

One of the methods that greatly assisted in this challenge was the aptly named *sub_handy_mask*. This routine takes a long data string to mask, a bit starting position, and a size. It returns back the masked portion of the bit string so that it can be compared or used elsewhere.

Another thing that greatly reduced our redundancy was a set of methods that could build output. For example, *sub_build_hex_string* takes a number to output, the size (nibble, byte, word, long), and whether or not it should print a '$' to signify a hex value.

The goal for the entire project was < 1500 lines of code. Through our techniques we were able to get it to < 1000 lines of code for all of the required opcodes, and < 1300 lines of code to handle more than 85 opcodes.

### *Extensiblity Goals*

*Look to the SDL section for a formal explanation.*

Another primary goal for us was extensiblity. We wanted to have a way to easily add more supported opcodes to the disassembler, without having to write a lot of code. By developing a language to describe operation codes, we were able to achieve this.

By describing our operations this way, debugging, testing, and adding new codes was extremely easy. Except for a few cases, we didin't need to add any additional code to describe an opcode.

## Program Flow

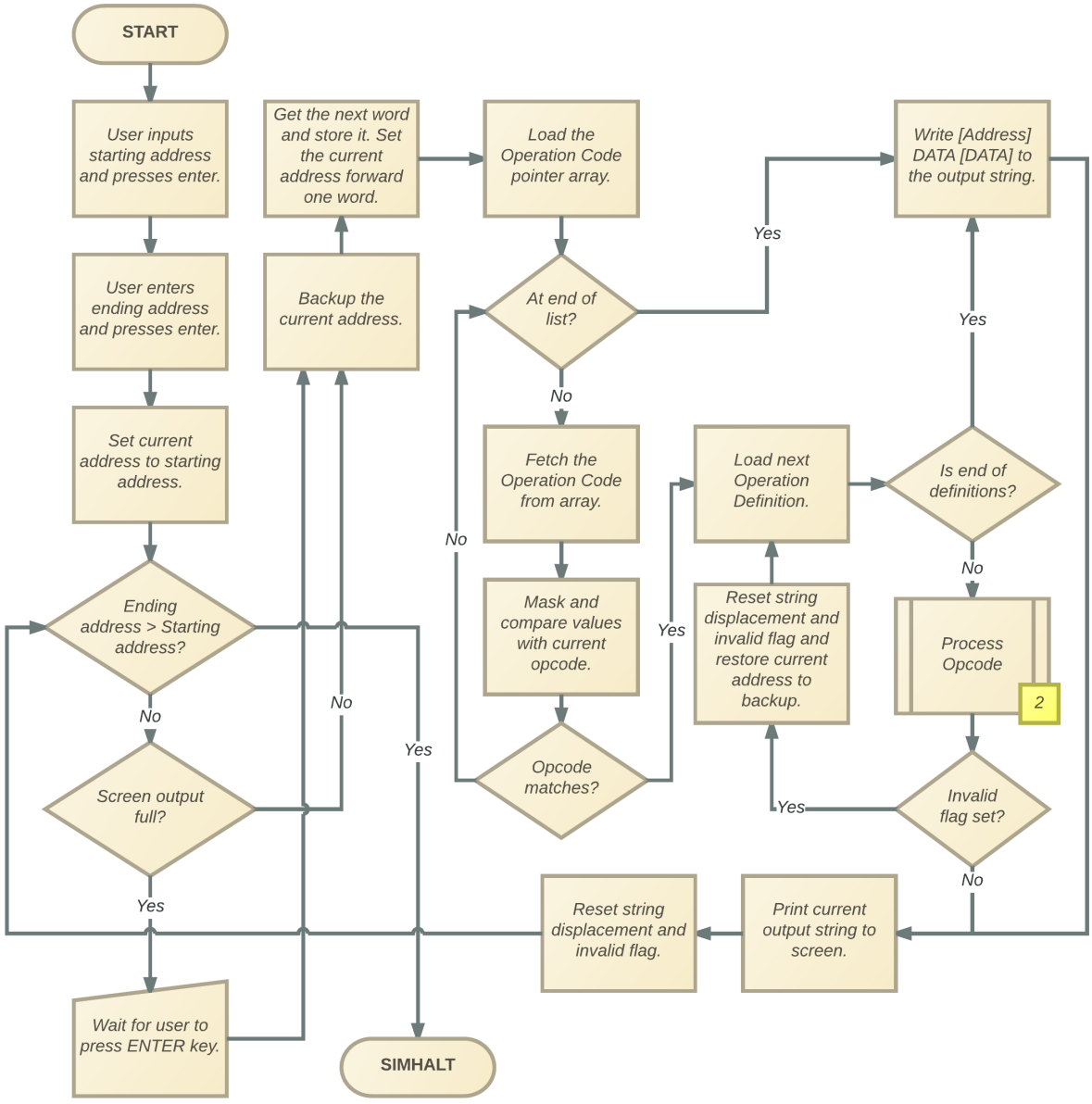
The program works by taking an opcode from the current address, masking uniquely identifying bits, and comparing it against a value that defines an opcode. If there is a match, it uses a prebuilt definition to try and output a string. If at any point there is an invalid bit or mode, an invalid op flag is turned on to signify the error.

From this point, the opcode is tried with another definition if it's available. If that still doesn't work, it goes back and continues to try other opcodes and definitions until it finds a match, or ultimately gets

output as data if no match is found. Figure 1 shows the main flow and Figure 2 shows the parsing operation.

## Figure 1: Main Program Flow

The main program flow reads in the opcode (or data) at each input and processes it. See Figure 2 for a breakdown of “Process Opcode”.



## Figure 2: Process Opcode Flow

Each opcode is parsed as described in the “Structured Disassembler Language” section.