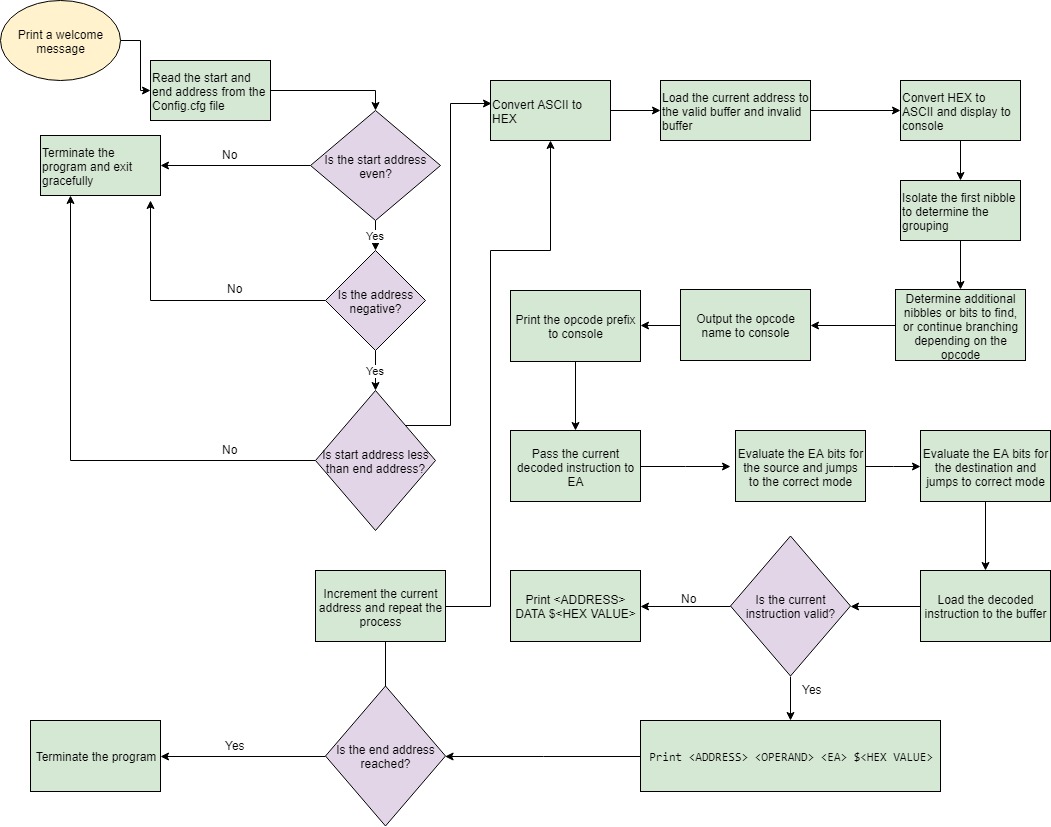
CSS 422: Hardware and Computer Organization

Disassembler – Project Description

Team – Single Precision REEE

## Program Description

Our team divided the project into three roles: input/output (IO), opcode (OP), and effective address role (EA). The team researched and designed the overall schema for the disassembler. **Figure 1** demonstrates the overall approach of the program:



**Figure 1: Disassembler Execution Flowchart**

### IO Program Flow and Structure

This disassembler is designed to read input from a Config.cfg file that contains a valid start and end address in which the user would like to have disassemble opcode instructions. Note that the ending address must be greater than the starting address. The program handles edge cases such as entering a negative or odd address to the Config.cfg file. After validating the starting and ending address, they will be stored in memory in address registers A5, and A6 to be used when decoding the instruction and translating the effective address.

### Opcode Program Flow and Structure

The opcode section of disassembler recognizes all the opcodes and then loads the next opcode in memory. The memory address of the recognized opcode gets stored in a the valid global buffer (**G\_BUFFER**) if it is valid or it will be stored in the invalid buffer (**B\_BUFFER**) if it is invalid. The validity is determined by a constant **IS\_VALID** which is set to 0 for and invalid opcode and 1 for a valid opcode.The program evaluates each opcode until all opcodes are uniquely and correctly identified. **Figure 2** highlights the core logic used to evaluate the opcodes in our disassembler program.

## 

**Figure 2: Disassembler Opcode Flowchart**

From the flow chart in **Figure 2**, the main routine **JSR\_OP\_ROUTINE** is run to identify a nibble value through a 12-bit bitshift. Once a nibble is identified, the respective routine is run. For example, if nibble **$4** was identified, then the routine **GROUP\_4** is run in response. Each group contains every possible opcode that is listed in the 68K for nibble grouping and initiates Callee-saving for the utilized registers.

After the group is identified, every specific code in that nibble group is now checked through specific bit masking or through a bitshift first into a bitmask. For example, if **$4E71** or **NOP** is the passed in **CUR\_OP\_CODE**, then it would be identified as **GROUP\_4**, and head through each specific **GROUP\_4** opcode check with bitmask. Eventually it will hit the routine **check\_OP\_NOP** which will do the appropriate bitmask comparison of **$4E71**.

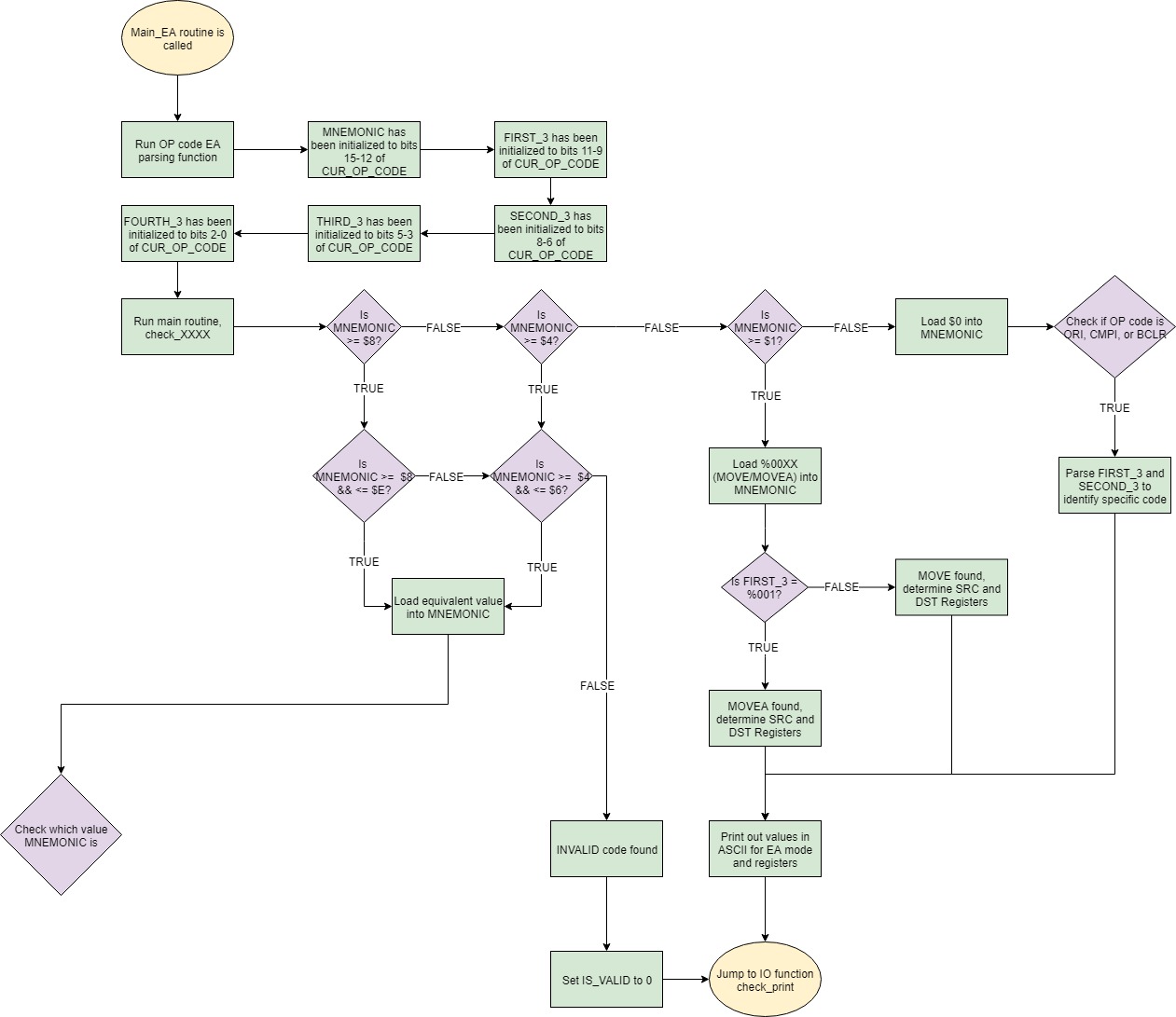
Once the appropriate bitmask comparison has been made, there are four potential options. The code either branches to **PRINT\_OP\_NAME**, **GROUPX\_SUCCESS**, **GROUPX\_INVALID**, or branches to the next checked opcode.

**NAME** refers to the name of opcode instruction, for example for **NOP** it would be **PRINT\_OP\_NOP**. During the **PRINT\_OP\_NOP** routine, first a printing routine is called to load the current opcode to the buffer. Then a size parsing routine that is respective to the command, most commonly **FIND\_2\_BIT\_SIZE\_7\_TO\_6**, is run and attempts to find the dynamic size field and set the constant **OP\_SIZE** to the recognized value. For example, the opcode **ORI** has a dynamic size marker in the bits 7-6 where it could be a byte, word, or long signified as **ORI.B, ORI.W, ORI.L** respectively. The size parsing routine will find the respective size through comparisons to **OP\_SIZE** and then add the appropriate “.SIZE” to the opcode in the buffer.

**X** refers to the identified nibble value (such as **GROUP4\_SUCCESS**). If the code branches to **GROUPX\_INVALID** it will then jump to a routine termed **INVALID\_OP**, which sets the global validity constant **IS\_VALID** to 0. After it will run **GROUPX\_SUCCESS** which simply restores the Callee saved registers and returns to the main routine of **OP\_JSR\_ROUTINE**.

### EA Program Flow and Structure

Variables such as **ADDRESS**, **DATA**, and **PTHSIS\_OPEN**, represent the ASCII values for what the variable name indicates in hex. For example, the variable **PLUS\_SIGN** has value the $2B which is equivalent to + in ASCII. Besides the ASCII variables, there are essential operative-variables such as **TRAILING\_SIZE**, **MNEMONIC**, **FIRST\_3**, **SECOND\_3**, **THIRD\_3**, and **FOURTH\_3**. **TRAILING\_SIZE** variable represents how many *bytes* I should consume to load the immediate/absolute short/absolute long value. **MNEMONIC** variable is the first four bits of the opcode that represents the family of instructions. **FIRST\_3** to **FOURTH\_3** variables represent the groups of three bits that appear first, second, third, and fourth. The EA algorithm heavily depends on the **TRAILING\_SIZE**, **MNEMONIC**, **FIRST\_3**, **SECOND\_3**, **THIRD\_3**, and **FOURTH\_3** variables. For example, it would refer to **FIRST\_3** for source register, **SECOND\_3** for source mode, **THIRD\_3** for destination mode, and **FOURTH\_3** for destination register. That was the most general explanation of the EA part. **Figure 3** demonstrates how the effective addressing modes get translated after they are passed from the opcode instruction.



**Figure 3: Effective Addressing Mode Flowchart**