# T34 Assembler

CENG 325 PROGRAM #2

Hunter, Daniel S. - SDSMT Student

# 1 CONTENTS

# 2 BUILDING AND RUNNING

Requirements

1. Python 3.9.7+ (May run on older versions but was not tested)

Running the program

```
python .\main.py <file>
```

Example

```
python .\main.py .\sample_1.s
```

# 3 MODULES

re – Python RegEx (Built-in)

os – Python Miscellaneous Operating System Interfaces (Built-in)

sys – Python System-specific Parameters and Functions (Built-in)

# 4 FUNCTIONS

All the following functions are a part of the t34Assembler class in t34Assembler.py.

**__init__(self)**

Constructor for the T34 Assembler class. Will set default values for all required variables.

**__hex_twos_dec(self, hex)**

Will return the negative twos complement decimal value from a given *hex* string.

**__twos_hex(self, val, nbits)**

Will return the twos complement hexadecimal value from a given decimal *val* with *nbits* of precision.

**__inc_pc(self, amount)**

Increment the program counter that was set initially as 0 from the constructor by *amount*.

**__reset_pc(self)**

Simply reset the program counter to 0.

**__number_format(self, nstr)**

Will convert an operand string *nstr* to a generally accepted hex string format that other functions within the class expect. It can handle binary, octal, decimal, and hexadecimal formats. Strings are not supported. The result of this conversion is returned, unsupported types return None.

**__string_format(self, hstr)**

Replaces hex strings *hstr* from the format that Python automatically puts them in with the hex() function to the format the program uses. Return the result.

**__extract_address(self, instr, operand, lineNumber)**

Uses the instruction *instr*, *operand*, and current *lineNumber* that the assembler is at to extract the proper hexadecimal address that will be placed in the bytecode. Examples include branch instructions that use symbols as an operand. This reads the hex and extracts correct twos complement bytecode if needed. Return the result.

**__get_addressing_mode(self, instruction, operand)**

Using the syntax of the *instruction* and *operand* along with RegEx queries, determine the addressing mode and return it.

**__read_format(self, line)**

Given a line of source T34 assembly code, read the format of the line and split it into 3 variables that will be returned in the following order: label, instruction, operand.

**__do_operations(self, operand)**

Determine if any arithmetic or logical operations are required for the *operand*, apply them, and return the result of the operations. This supports addition, subtraction, division, multiplication, exclusive OR, OR, and the AND operation.

**__replace_symbols(self, operand)**

Using the *operand* check the symbol table for values that are other symbols and replace them with the uppercase hexadecimal address value of those symbol references. Return the replaced value.

**__calc_bytes(self)**

Iterate the object code and calculate the total bytes, storing it in self.bytes.

**__xor_previous_bytes(self)**

Iterate the object code and take the XOR of all bytes to get the hexadecimal checksum and return it.

**__add_symbol(self, label, operand, lineNumber)**

Use the *label* to determine duplicates in the symbol table, if there are no issues, map the label to the *operand* so that it is a valid symbol within the symbol table. Error checks will use the *lineNumber* to give a precise location of where it occurred. Return True for success and False to determine an error occurred.

**__assemble(self)**

Acting as the first pass of the assembler, iterate the source T34 Assembly code to generate a symbol table and valid PC for every line of the program. Handles jumping and ORG instructions directly. Opcode, operand, memory full and duplicate symbol errors can occur in this phase.

**__assembler_print(self)**

Acting as the second pass of the assembler, generates user output while also handling the generation of the object code. Handles utilizing the symbol table and OPCODE maps for object code generation while directly handling the CHK pseudo instruction. Additional error checking will occur in this phase.

**__symbol_print(self)**

Display the symbol table to the console in alphabetical and numerical order respectively.

**fopen(self, path)**

Using a *path* to a file, open a file handle that is stored in the class variable *self.file*.

**fread(self)**

Read all the lines of the opened file handle and store the lines in the class variable *self.source*.

**fwrite(self, path)**

Write the object code to a file located at *path*.

**getSymbols(self)**

Return the symbol table contents.

**getCode(self)**

Return the current generated object code.

**fclose(self)**

Close the file handle containing the source assembly code.

## 5 TESTING

Most of the testing was done manually with the use of Windows calculator using the Programmer mode. Output matching was done visually comparing the included test cases while base conversion was done with the calculator. Twos complement was checked with manually written and programmatic calculations as well as with the help of RapidTable’s hexadecimal to decimal converter located at https://www.rapidtables.com/convert/number/hex-to-decimal.html.

Some additional notes will be listed in the next section to explain test cases that I did not extensively check for as well as unsupported operations. Some unsupported operations may break the program with test cases I did not properly anticipate.

## 6 Additional Notes

I was unable to get HI-byte and LO-byte number formats, present address identifier using an asterisk, and storing strings in labels to work properly with this version of the assembler. I would need to further optimize the main assembler functions to handle these test cases that I did not catch as a requirement in time.

Additionally, bad branches do not generate object code for the invalid address reference.