**Assembler Writing Project**

We received an imaginary processor and an imaginary assembly language. Our task was to take a file containing assembly commands and create machine language files that the computer understands.

Each instruction in the system had to be encoded into 32 bits (4 bytes) using the little-endian method (bits 0-7 in the byte at the lowest address). Every instruction in the language had an opcode identifying the instruction, and some instructions also had a secondary identifier called funct.

In the assembly language, there were two types of statements: directives and instructions.

1. **Directive Statement:** Directs the assembler on what to do when operating on the source program, such as memory allocation or initializing variables in the program.
2. **Instruction Statement:** Produces the encoding of machine instructions for execution during the program's runtime.

We were required to write a two-pass assembler because the assembler translates the assembly file line by line. If, for example, on line 5, we needed to load the address of a certain label that only appears on line 15, it would be impossible to fully encode that line since we wouldn’t know the label’s address on the first pass. Therefore, in the first pass, I performed partial encoding and built a symbol table. The symbol table included the label's name and address so that in the second pass, we could identify the label’s address and complete the remaining encoding.

One of the requirements was to separate instructions and data so that the instructions would appear first in the machine language encoding, followed by the data. Additionally, the assembler had to output error messages for each line where a specific error was detected, along with a description of the error.

During the first pass, I performed precise parsing on each instruction. I checked the type of each instruction and, based on that, knew which function to call to check the instruction. If there was an error in the instruction, I didn’t stop the program but instead printed the error details and the line where it occurred, then continued to the next instruction so that the user would know all the errors that needed to be corrected in the assembly code before running it again.

I used several structures to build the assembler:

1. **Symbol Table:** This table was built as a dynamic linked list. Each node had the label’s name, its address in memory, and its type. In the second pass, we could refer to the symbol table for an unidentified label and perform a linear search to find its address for encoding purposes.
2. **Data Table:** This table was also built as a linked list, where each node had its number in memory, the type of instruction, and its binary encoding. The data table was fully encoded in the first pass.
3. **Instruction Table:** The instruction table was also a dynamic linked list. Each node had its number in memory, the type of instruction, and its partial/full binary encoding. Since some instructions included symbols, the completion of the instruction table was only done after the second pass.
4. **Combined Table:** This table was stored as a static array because I knew the size of each list, so there was no need for a dynamic structure. First, I inserted the entire instruction table, then the entire data table, since according to the definitions, the instructions’ encoding should precede the data.

**Note:** Since I completed the project in my second semester before I studied the Data Structures course, I used a linked list for the symbol table. After gaining additional knowledge and if I were to do the project today, I would use a hash table for the symbol table so that searching for a symbol could be done in constant time instead of linear time in a linked list.

The final product of the project consists of three files:

1. **Object File:** A file containing the machine code in hexadecimal.
2. **Externals File:** A file containing details about all the addresses in the machine code where a symbol declared as external (a symbol that appeared as an operand in the extern directive) is encoded.
3. **Entries File:** A file containing details about every symbol defined as an entry point (a symbol that appeared as an operand in the entry directive).