Assembler Project Final Report

Group: Ethan Netsch

**Intro**

My project was written with **C++.** Out of the 6 levels, I mostly completed the first 5 out of 6, excluding macros.

The core processing of my assembler is based on the following design patterns:

1. **Command Pattern -** Encapsulating a request or a command as an object so that it can be processed at a later time
2. **Factory Pattern** - Creating a class that handles the creation of objects without the need for the user to specify their types

**Core Assembler Features -**

**Instruction Creation**

In the case of the assembler, I considered instructions themselves to be commands and mainly work with Instruction Objects. This was useful for the design of the assembler because it utilizes a two pass algorithm to assemble the code.

The instruction object is an instance of an implementation of the Instruction interface. This interface defines the following methods: **assemble** - return an integer representation of the assembled instruction, **length** - return the number of bytes of the instruction as an integer, and **address** - return the address of the instruction as an integer. I chose to use an interface so the actual assembler only needs to work with an instruction pointer and doesn’t care what format or actual instruction it is working with. All the assembler needs to know is that it is working with an instruction and for any given instruction it should be able to assemble it, retrieve its length, and get its address.

The classes that implement this interface are the ones that differ in their assembly process - **Format1\_Instruction**, **Format2\_Instruction**, **Format3\_Instruction**, **Format4\_Instruction**, **RSUB**, **BYTE**, and **WORD**.

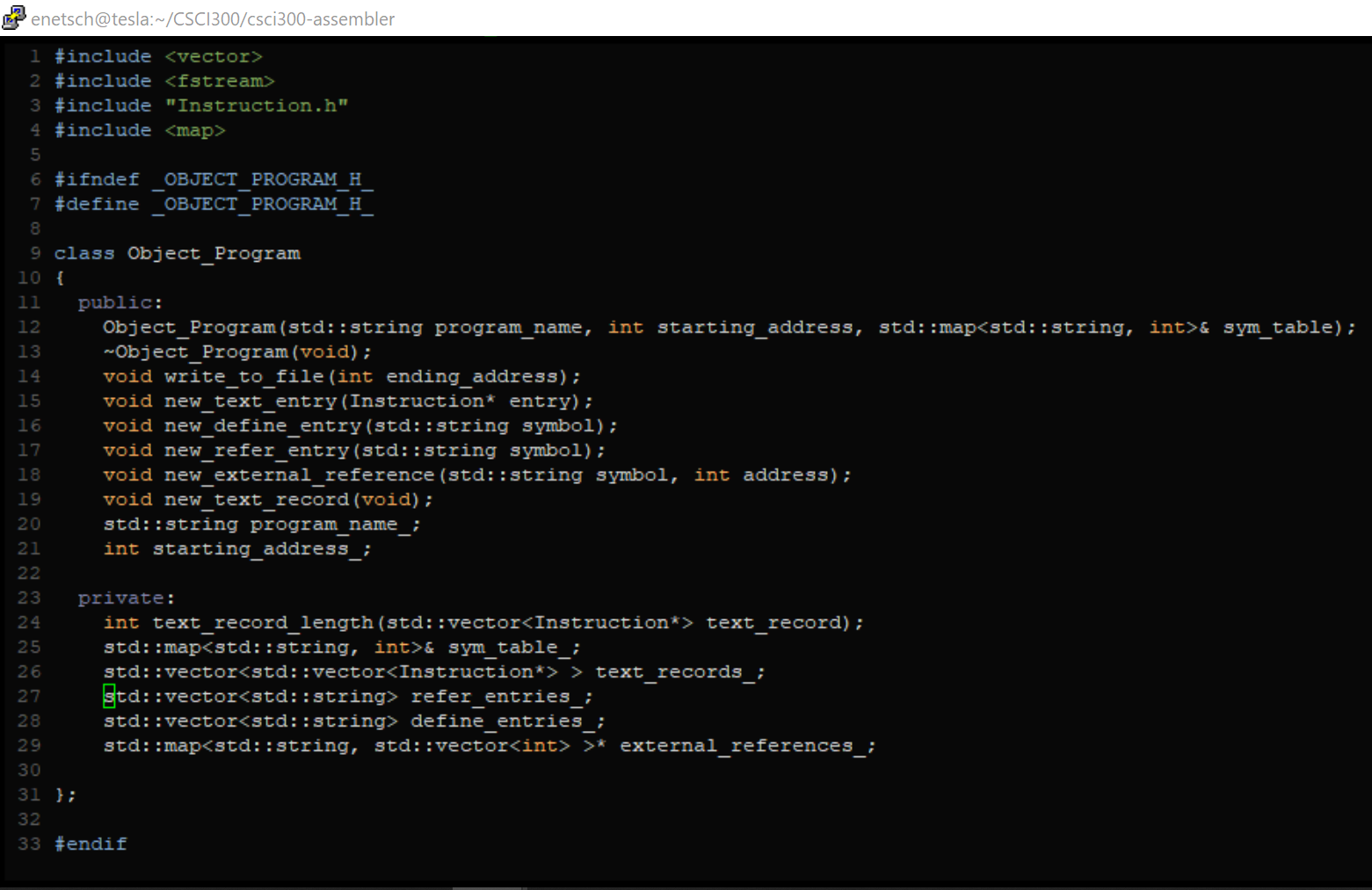
(Here Instruction refers to anything that generates object code)

In order to construct one of these objects, I made a factory class **Instruction\_Factory,** that contains pointers and references to all assembler state variables. The method create\_instruction can be called with the instruction name and its operands and correctly construct a new Instruction of the correct type. In order to determine the correct Instruction type to assemble, mnemonic instructions are stored in 4 different optables. When an instruction name is found, it calls the corresponding Instruction type constructor for the given op table. For example, the instruction ADDR is stored in **optable\_2**. If **create\_Instruction** is called with a parameter ‘ADDR’, the Format2\_Instruction constructor will be called with the parameters: **pc**, **opcode**, and **operands**. For Format3 and Format4 instructions, a reference to the symtable is passed to its constructor so that it can look up addresses when its assemble method is called.

**Object Program Generation**

In order to generate an object program from a given input file, I created a class **Object\_Program** thatmaintains multiple data structures(maps, and vectors) for: text records, refer records, define records, and modification records. This class essentially organizes these records while the assembler processes the input file. It implements methods that will store the records until the **write\_to\_file** method is called. When this method is called, an output file is generated that will write all of this information out.

The Object\_Program class:



**Main Loop**

With the **Instruction\_Factory**, **Object\_Program**, and **Instruction** classes, the main loop of the basic assembler is rather simple. This logic lives in a file - **driver.cpp.** Here the assembly code and assembler directives are processed. In pseudocode it works in the following way:

1. Read input from the command line for an input file name
2. Create:
   1. **symtable**: map<string, int>
   2. **instruction\_factory**: Instruction\_Factory
   3. **loc\_ctr** : int
   4. **base**: string
   5. **instructions**: vector<Instruction\*>
   6. **object\_program** = Object\_Program\* -> name = “NONAME”, starting\_address = 0, sym\_table = sym\_table
3. Open input file and begin parsing line by line
4. **Pass 1** - For every line,
   1. separate by whitespace into 3 tokens: **label**, **command**, **operands**
   2. If **command** = “START”
      * 1. Set object\_program starting\_address = operands
        2. Set object\_program name = label
   3. If **command** = “BASE”
      1. set **base** = command
   4. Else if **command** = “RESW”
      1. Increment **loc\_ctr** by 3\***operands**
   5. Else if **command** = “RESB”
      1. Increment **loc\_ctr** by **operands**
   6. else
      1. Call **create\_instruction**(command, operands) store as **instruction**
      2. If there is a value for label, add to symtable: **symtable[label] = loc\_ctr**
      3. Increment **loc\_ctr** by **instruction.length()**
      4. Add **instruction** to **instructions**
      5. Create\_new\_text\_entry on object\_program
5. **Pass 2 -** For every instruction in **instructions**
   1. Write **instruction.assemble()** to console
   2. **Object\_program write\_to\_file**

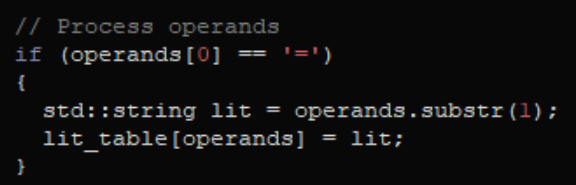
**Literals**

Implementing Literals was again relatively simple within my current design. In order to do so I had to add a new data structure to my driver:

Create :

**lit\_table** : map<string, string>

In order to populate this table, a simple check was added to see if an instruction’s operand was a literal. If it was, the literal was added without the = to the lit\_table with key = to the literal.



For example, the instruction “**ENFIL LDA =C’EOF’**”, will be added to the lit\_table as:

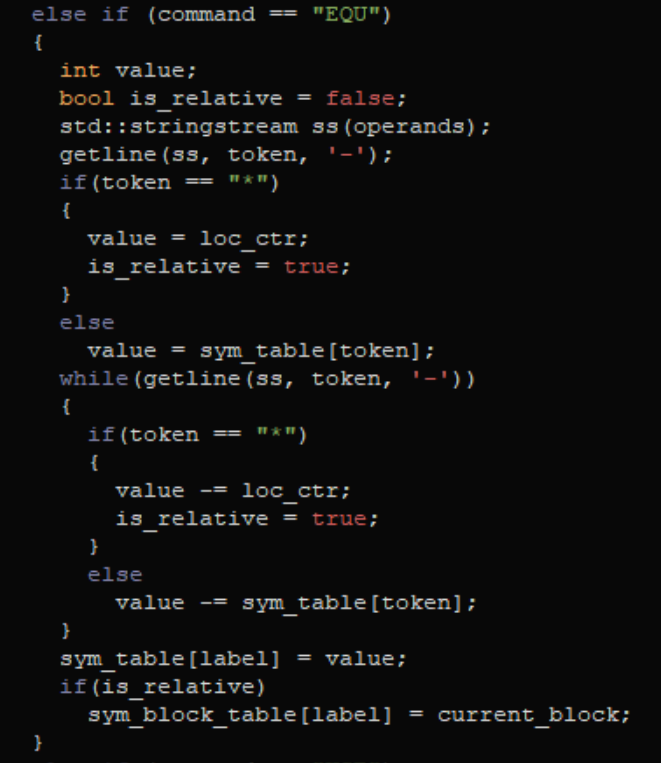
**lit\_table[“=C’EOF’”] = “C’EOF’”.**

In order to actually write out the literals to the **object\_program**, another check was added for when the command was equal to the string “**LTORG**” or “**END**”. When this occurs, the literals are dumped from the lit\_table with the following process:

1. For every entry in **lit\_table (key=literal, value=value)**
   1. Add to the **sym\_table [literal] = loc\_ctr**;
   2. Call **instruction\_factory** **create\_instruction**(“BYTE”, **value**) = **instruction**
   3. Call **object\_program new\_text\_entry(instruction)**
   4. Increment **loc\_ctr** by **instruction.length()**
2. Erase all literals from lit\_table

**Symbol Defining Statements**

I added this functionality only to when the **EQU** assembler directive is called. However, I would have liked to implement this functionality to all operands fields. Additionally, the only operator my implementation supports is -. This feature did not require any additional data structures and just required parsing the operands and looking up its tokens in the symtable.



**Program Blocks**

This is the section that I struggled the most with when implementing my assembler. Because program blocks require the assembler to keep track of which block symbols were defined in, my current design was relatively naive. When creating an instruction or a symbol in my design, the assembler assumes addresses are absolute. However this is not the case of program blocks and caused many difficult issues in my design.

An ideal way to implement this feature would be to completely refactor my **sym\_table** and **Instruction Interface.** Both of these structures would not only need to keep track of the block in which they are defined, but also have a way to look up the actual starting address of the block when it is time to be assembled. Due to time restraints, I was unable to complete this refactor. Instead, I approached this problem with more of a brute force mindset.

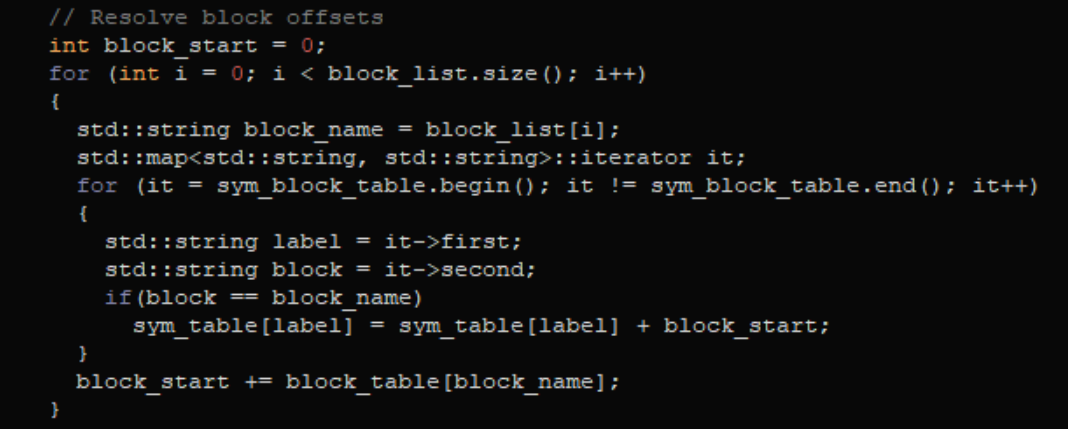
In order to implement program blocks, i introduced 3 new data structures and a string:

* block\_table: map<string, int>
* sym\_block\_table: map<string, string>
* block\_list: vector<string>
* current\_block: string

All of my handling of program blocks required maintaining these structures in parallel, which is not a very good design and can cause many errors.

With the following changes, most of the program block functionality is completed. However, one feature that is missing is updating the start address in text records to reflect the offset of the corresponding program\_block.

1. If **command** = “**USE**”:
   1. save the value of the **loc\_ctr** in **block\_table** at key **current\_block**
   2. Set the **current\_block** = to **operands**
   3. if **current\_block** not in **block\_table**
      1. push **current\_block** onto end of **block\_list**
      2. Add entry to **block\_table** with key = **current\_block** and value = 0
   4. Set **loc\_ctr = block\_table[current\_block]**
   5. Call **object\_program** **new\_text\_record()**
2. If **command** = “**END**” -> **block\_table[current\_block] = loc\_ctr** (Update the last blocks length in **block\_table** before ending)
3. Whenever adding any symbol to the **sym\_table**,
   1. Also add symbol and **curent\_block** to **sym\_block\_table** if value is relative
4. Before pass 2,
   1. Calculate the actual starting address of all blocks in the order of **block\_list**
   2. For every symbol in **sym\_block\_table**
      1. Find symbol in **sym\_table**
      2. Increment value in **sym\_table** by symbol’s block’s starting address



**Control Sections**

Much like my problems with program blocks, some of my existing data structures didn’t easily allow the implementation of control sections. However, I feel that the design I came up with for control sections is much more elegant than my program block solution.

The only new data structure I had to implement in **driver** to utilize control sections was:

* **external\_symbols** : vector<string>

However, **Object\_Program** required 3 new data structures and methods for adding to them:

* **refer\_entries\_** : vector<string> -> new\_refer\_entry(string symbol)
* **define\_entries\_** : vector<string> -> new\_define\_entry(string symbol)
* **external\_references\_** : map<string, vector<int>>\* -> new\_external\_reference(string symbol, int address)

These structures are utilized with **Object\_Program** to generate define, refer, and modification records when **write\_to\_file** is called.

In order to communicate to the **object\_program** that these symbols are being read, I made the following changes:

1. If **command** = “**EXTDEF**”
   1. Tokenize operands, and call **new\_define\_entry** for every token
2. If **command** = “**EXTREF**”
   1. Tokenize operands by ‘,’,
      1. call **new\_refer\_entry**(token)
      2. Add token to **sym\_table** with value 0 (all address fields for external references should be 0)
      3. Push token onto **external\_symbols**

Additionally, I created a method **check\_external\_references** that will tokenize a string and see if any of its tokens are in **external\_symbols,** if so call new\_external\_reference for the given token and the lines address.

The most important part of this implementation of control sections is when **command** = “**CSECT**”. When this occurs, **object\_program write\_to\_file** is called to write out the current control section’s object code. Afterwards, all data structures previously defined are reset to their original values. By doing so, a new control section can be started with blank state.

**Conclusions**

Although I found this project to be extremely difficult, it was also very rewarding. Not only did I learn how to build an assembler, I feel that I also greatly increased my C++ programming skills and my knowledge of software design. This was the largest project that I have ever worked on and designing it in a way to be scalable was what I struggled with the most. Through this process I learned the importance of future proofing your software so that you don’t run into problems like I did when trying to add program blocks and control sections.

If I could do this over again, I would start the project with a much more complex symtable where more data could be more easily added. Because my symtable only included the symbol name and its address, it made it very difficult to handle any relative addressing.