CmpE 230 Project 1- Assembly Interpreter
Students: Hazar ÇAKIR – Doğukan AKAR

Briefing:

We did this project as 2-students group. We use C++ to program Assembly interpreter. We wrote our code in a way that it can run in both UNIX and Windows. One can compile the project with G++. It takes the path of the ".asm" file as argument and shows results in console. It works well for all test-cases that are given and we think that for any other cases. Our program is written to execute every valid assembly code in given restriction in project description.

Code Analysis:

Source code is well commented. Every function and variable have names according to their function. Even so, we will examine our code elaborately. In order to handle errors, we used try-catch structure in whole project. Whenever an error occurs, we threw "Error".

We store all components of processor in global variables according to their real size. We chose to use unsigned short to store "word" size variables and unsigned char to store "byte" size variable because unsigned short stores 16-bit data and unsigned char stores 8-bit data as the same as processor. We used a char array sized 2^{16} to represent 64 kb memory. Flags are represented as bool values. 8-bit registers are defined with pointers pointing corresponding register. We used **little Endian** representation in both registers and arrays because C++ uses it naturally. We store assembly code in the vector<string> lines. In the first iteration of folder, the variables stored in vector<string> varLines, after first iteration done, in the main with defineVars(), variables will be placed to the memory array. We use maps in order to ease our job. There are two maps for registers. One of it for 16-bit registers where other one is for 8-bit registers. They match name of the register which is string with pointer of the register. There are also 2 other maps to keep variables and labels. Map<string,pair<char, int>> vars matches names of the variables and their type-position pair. The other one is map<string, int> labels which matches names of label with their place as number of line.

There are a lot of helper functions. Let us examine first the most crucial ones:

The most important functions are the functions that are represents assembly functions. We have a lot of functions such as add_func(), sub_func()… Our template for these functions was <functionName>_func(parameters). For every function there are two different version: One for "word-sized" operation, another for "byte-sized" operation. As we mention before, word-size operations take arguments as unsigned short where byte-size operations take arguments as unsigned char. There are two types of parameters: "Pointer"-ones and "Values". All functions take these two types of parameters, for example add_func() takes a pointer and a value; cmp_func() takes two value; mul_func() takes a value and so on. It is assumed that parameters are valid, so there is no control for correctness of inputs of functions. Some of the functions uses helper functions like rcl_func() and int21h_func(). They do not return any value except Jump functions. Jump functions return bool values whether the jump operation will be carried out or not.

The controls of parameters are done in main function. In main function, first readFile() and defineVars() functions are called. They parse the file in the vectors and then allocate variables. Tokenization and other parsing staff is done in these functions. Then we start to traverse lines vector with cursor variable. Every element of lines vector represents one line and one function. In every iteration, one operation is done. There is vector<string> args to store arguments of corresponding line. takeArgs() function is to get arguments separately. After this point, the function defined with if statement and for each function, corresponding controls are done. There are several helper functions to deal control process. getType() function retrieves the type of the arguments as 'w', 'b' or 1: 'w' for word, 'b' for byte and 1 for unknown type. The type feature makes our control work very easier. In this method, the arguments are checked if they are valid arguments or not. If argument is in a form that carries no type, then it is an error. The actual control mechanism is the function interpretValue() which is called inside this method and also several places. interpretValue() function is analyze the data given and return the value of that data as integer. In this method, the arguments interpreted from hex form, decimal form, or char form. If none of these is possible, then throw an "Error". There are used a lot of helper method in this function, but you can understand them when you read the code.

According to types of the arguments, necessary controls are done for each specific function. After passing size matching process, there is one more step to call our assembly functions. Getting value or getting pointer of the argument. As we know, our functions have two types of parameter. So in this step, our arguments are

transformed into corresponding forms. getPointer() and getValue() (and also getPointer8() and getValue8() for byte-type operations) functions are used for this purpose. After that point, our functions are called. For jump operations, cursor changed to corresponding line value.

Completion Status:

Project is done completely. There exist no bugs or missing parts encountered in test-cases.

Prepared by:

Hazar ÇAKIR – Doğukan AKAR

2017400093 --- 2017400003