Lynne Coblammers
EECS 665
Lab 8 Report
2015.11.11

The purpose of this lab was to create assembly code from intermediate code. We were provided with a main function that called several test functions which performed various calculations and a function call. This file was compiled using gcc. We were also provided with the implementation of those functions, which were written in intermediate code in the form of quadruples. A lexer and parser for this intermediate code were provided to us. The parser had actions embedded in the instructions for much of the code, including add, but the goal of this lab was to fill in actions for the remaining expressions, including subtractions, multiplication, division, modulo, left and right shifts, arguments, and calls. These actions needed to print out the correct assembly code to execute those instructions. Once these were filled in, the generated assembly was linked in to provide definitions for the test functions called from main, and the implementation could be tested by running the executable.

The assembly code for subtract and multiply was very similar to add. Both loaded one argument into the eax register using movl, and then called the appropriate arithmetic operator on the other argument and eax. Division and modulo were both a little more complicated. After loading the first argument into eax, it needed to be sign extended to essentially clear out the value of edx. For division, the division instruction could then be executed using the other argument, leaving the result of division in eax. Modulo also uses division, but the remainder of division is left in edx, so another step is required to move the contents of edx into eax. To carry out the shifts, the argument to be shifted was loaded into eax, and the amount to shift by was loaded into ecx. The shift instructions require that the amount to shift by is an 8 bit register, so shift was carried out on cl (the lowest 8 bits of ecx) and eax. To load arguments for a function call (IARG), each time an argument was seen, it was pushed onto the stack using the pushl instruction. Finally, to call a function, call was called with the function label. After returning from the function, the stack pointer needed to be updated by the number of arguments that had been pushed on the stack, so an addition was performed on esp, adding the number of arguments times 4.

There were a few obstacles along the way. The first was that the order of the arguments was reversed from the expectation. For example, in the expression ID SUB ID, when subtraction is called on (12,6), the first ID actually contains 6 and the second contains 12. This meant division and subtraction were not performing as anticipated initially. Another challenge was in adding the arguments. Initially, I was using function_labeltemp to get the value to push on the stack. However, this meant that the string label (e.g. strval0) was printed without a $, which caused problems in actually passing the string to printf. This was switched to function_printtemp, which appended a $ to the argument (e.g. $strval0). That allowed the label to be resolved to its actual value when being added to the stack. The final challenge was in the call. Initially, I had just added an assembly instruction to call the function label. However, after calling and executing the function, I was seeing a segmentation fault. This is because I hadn't updated the stack pointer after returning from the call. After looking at the assembly generated from test.c, I could see that I needed to add to the stack pointer based on the number of arguments I had pushed onto the stack. Once this was resolved, the program performed correctly for all of the test functions.