Computer Systems, Spring 2022

Assessed Exercise

Assembly Language Programming

In this exercise you will write and test an assembly language program on the Sigma16 architecture. The program illustrates a variety of basic techniques, including conditionals, loops, procedures, arrays, linked lists, and I/O requests. The aims of the exercise are to

- Gain experience with machine and assembly language.
- Develop a deeper understanding of high level programming language constructs by studying their implementation on a computer architecture.
- Practice the implementation of arrays and lists.
- Improve your general programming skills by learning to use a systematic programming methodology.
- Learn how to read a program and modify it.

The systematic programming methodology is a central aspect of the exercise. The program is specified at three levels:

1. High level language notation, using while loops, if-then-else, etc.
2. Low level language notation, where you have assignment statements, write, goto L, and if condition then goto L. Note that in the low level language, the only thing allowed after "if condition" is a goto statement.
3. Assembly language implementation.

It is essential to develop the program systematically, going through each stage. If you just start by writing assembly instructions, it's unlikely that you will get the program to work. The *process* of developing the program is more important than the program itself.

You are given two documents:

- This document, describing what to do.
- A file `OrderedListsEXERCISE.asm.txt` that contains a partial implementation of the program, including the high level language version for everything, the low level notation for some parts, and also the assembly language for some parts. The file contains most of the solution to the problem, and it also provides a lot of programming examples that you should study in detail.

You may work on the program both during the scheduled labs (where there will be demonstrator/tutor support) and outside scheduled labs (without support). You are encouraged to discuss your work with your demonstrator/tutor. You may discuss the general approach to the exercise with other students, but **you are not allowed to get actual code from other students, or to give your code to others**. Your solution will be submitted on Moodle.

1 Specification: What the program does

There is an array named `list` containing `nlists` elements, where `nlists` is given the initial value of 5. Each element of the array is a header node pointing to a list of numbers in increasing order. Initially every element of the array is an empty list.

There is also an array of records, where each record describes a command to perform. The record has three fields: `code`, `i`, `x`. The meanings are as follows:

- code = 0: **terminate** the program.
- code = 1: **insert** x into list[i] so that the list remains ordered.
- code = 2: **delete** the first occurrence of x in list[i], if any. If x doesn't occur in the list, do nothing.
- code = 3. **Search** list[i] for x; if found print "Found", otherwise print "Not found".
- code = 4. **Print** the elements of list[i].

The task is to write a program that takes the array of commands and executes them. The EXERCISE program implements the building of the heap, the loop that traverses the array of commands, and two of the commands (terminate and insert). The program runs as it is, but three of the commands do nothing (print, search, delete). Your task is to implement those three commands. As you do so, you should study the existing code carefully; all the techniques you need are illustrated in the code that's provided.

2 How to proceed

The program should be developed systematically, following the programming guidelines presented in the lectures, just like the examples that have been given in lectures, in tutorial solutions, and in the EXERCISE file. *Follow the programming guidelines* and remember that it saves time to do the comments first.

1. Download OrderedListsEXERCISE.asm.txt from Moodle.
2. Read and study this document and the EXERCISE program. It is recommended that you spend the first lab session simply reading the program, studying the examples in it, and getting familiar with the program.
3. Find where the various algorithms are: the high level algorithms, the low level ones, and the assembly language implementations.
4. Modify the comments at the beginning of the program. The comments should give the name of the program, your name and matriculation number, the course (CS1S) and the date (March 2021).
5. As you progress, you should continually be reading this document and the EXERCISE file, which contains a lot of examples. Pay attention to the general style and layout of the EXERCISE file, as well as the algorithms it contains.

6. Run the EXERCISE program. It should terminate after executing around 9,200 instructions. It outputs a description of each command, nothing else. What was the computer doing as all those instructions executed? It was building the heap, traversing the set of commands, and executing the commands. Two of the commands are actually implemented (terminate and insert), and the insertions took some time. Why was there no output from the print commands? Because print (as well as search and delete) contain no code; they simply go back to the command loop.

7. Before doing any programming, it's useful to learn how to find a data structure in memory. Let's work out what is in list[0].

   (a) Look in the assembly listing for the data statements defining the array of lists (this is toward the end of the file, right before the "Input Data" section).

   (b) Get the address of list[0].next. Call it X1 (but you should find the actual address as a hex number). X1 points to the first node in the list. (Notice that "list" is an array of header nodes, not just an array of pointers. This makes the insert and delete operations easier to implement.)

   (c) Look in the memory display for address X1+1; this is the pointer to the first node for the list; let's call it X2.

   (d) Now look at the first node in the list, at address X2 (containing the value field, which is 006d) and X2+1 (containing the next field, which is X3). So the first element of the list is 006d. Draw a box for the node, write its address beside the box, and fill in the value and next fields. Draw an arrow from the list[0] to the node.

   (e) The next element in the list is at address X3, with value 0073 and next = nil. Again, draw a box, fill in the fields, and draw an arrow from the previous node to this one.

   (f) Since the next field in the last node is nil, the list ends there, and the complete list is hex [006d, 0073]. Try converting these hex numbers to decimal, and look at the insert commands to see if the result looks ok.

   (g) While working with lists, it's very important to draw diagrams; don't just work with text.

8. Now, work out the value of list[2] by examining the memory.

9. Locate the places that something is missing. These spots are all identified in the program with a comment like this, which indicates what you will need to write:

```
; *** EXERCISE Insert assembly language for CmdPrint here ***
```

10. Note that the file you are given implements the main program in full, and it also implements the insert command in full. Here is a summary of the missing pieces, which you should implement; it is recommended that you write these pieces in the order given:

(a) low level for CmdPrint

(b) assembly language for CmdPrint

(c) low level for CmdSearch

(d) assembly language for CmdSearch

(e) low level for CmdDelete

(f) assembly language for CmdDelete

(g) assembly language for ReleaseNode

11. Find the high level code for the print command; it begins with <CmdPrint>. Study this code. Try hand executing it on list[2]; you'll discover that the print high level code does exactly what we just did above in examining the memory to find the values in a list.

12. Translate the high level code for print to the low level form, and insert it into the file (look for "; *** EXERCISE Insert low level algorithm for CmdPrint here ***"). All you have to do is replace the while construct with suitable if-then-goto; the assignments and print statements are already low level.

13. Now translate the low level code for print to assembly language ("; *** EXERCISE Insert assembly language for CmdPrint here ***").

14. Test the program. Before running at full speed, it's a good idea to single step it and check that each instruction does the right thing. But the program will execute more than a thousand instructions before it encounters a print command. So you should use a *breakpoint*. That means you tell the emulator to stop when it reaches a specified point in the program.

15. To set a break point, first find the address of the label CmdPrint. You'll have to look at your assembly listing to find that. Suppose it's 01c8. You want the emulator to run at full speed until the pc register becomes 01c8. In the processor panel, click Breakpoint. In the window, enter *$01c8* (of course, use the right address for CmdPrint). Then click Refresh, click Enable, and click Close.

16. Now, click Boot and Run. The processor should stop when it reaches CmdPrint. Now you can single step through your program as it traverses the list.

17. When your implementation of print is working, run the entire program at full speed, and you'll see the results of all the print commands.

18. Now the entire program is working, apart from Search, Delete, and ReleaseNode.

19. Do the search command first. This is a loop that traverses the list, looking for an element whose value is x. The list traversal is quite similar to the traversal in print. Start with the high level algorithm for search, which will be similar to the high level algorithm for print, and then go through the usual process: low level algorithm and finally assembly language.

20. Then do the delete command, and finally implement ReleaseNode.

Here is the output from the finished program:

```
Insert     99 into list      2
Print list      2
    99
Insert    100 into list      4
Insert    115 into list      2
Print list      2
    99    115
Insert    107 into list      3
Insert    112 into list      2
Insert      1 into list      2
Print list      2
     1     99    112    115
Insert    120 into list      3
Insert     98 into list      2
Insert    109 into list      0
Print list      2
     1     98     99    112    115
Search list      2 for    112
Found
Delete    112 from list      2
Print list      2
     1     98     99    115
Insert    115 into list      0
Search list      2 for    112
Not found
Print list      0
   109    115
Print list      1

Print list      2
     1     98     99    115
Print list      3
   107    120
Print list      4
   100
Terminate
```

3 Hand in

Your program will be a single file named OrderedListsEXERCISE.asm.txt. It should contain:

- Initial comments identifying the program, and giving your name, matriculation number, the course, the date, and lab group.
- A short status report, which says what the status of your solution is. Is it finished? Does it assemble without errors? Does it run and does it produce the correct results? If you got stuck, what parts are missing or

what errors are you getting? If your program is working correctly, just say so — one sentence suffices. If there are problems, it is to your benefit to say as clearly as you can what is wrong — longer than just a sentence. The status report should be in the form of comments at the beginning of the file, right after the identification comments.

- The program, including the high level algorithm, the low level algorithm, and the assembly language code, following the guidelines given above.

Submit your solution on the Moodle page before the handin deadline. The program will marked out of 100:

- (5) Identifying information (in the form of comments at the beginning of the program). The first comments identify the program, giving your name, and saying what the program does. These may be the easiest marks you'll ever get: 5 marks for providing your name!
- (5) Your status report. State clearly whether the program works. If parts are not working, say so.
- (20) The Print command.
- (20) The Search command.
- (20) The Delete command.
- (10) ReleaseNode
- (20) The program assembles and executes correctly.

Many of the marks come from your documentation and your programming process. A well documented program that doesn't work may get a higher grade than a working program without comments. This is realistic; in the "real world" an undocumented program has little value. You can get a good grade by including the various required comments — and this will also help you to get the program working!