CIS 240 Fall 2020

Homework #9: Stack Attack {150 pts}

"A good programmer is someone who always looks both ways before crossing a one-way street." (Doug Linder)

Preamble:

In this assignment you are going to write a small compiler that will transform code written in a new stack oriented language, called J, into LC4 assembly code much the way that the lcc compiler converter converts C code into assembly. **Your compiler will be designed to use exactly the same calling convention that lcc does that way your J programs can call subroutines compiled by lcc and vice versa**.

Once again you are expected to do all of your work on the virtual machine provided using the clang compiler. That is where your code will be tested and graded. You are expected to provide all of the C source code and a makefile for producing the final executable program, jc. This program will act much like lcc does, when provided with a properly formatted source file in the J language it will produce the corresponding assembly code. That is

```
>>  ./jc foo.j
```

will produce a new file called foo.asm if foo.j contains an acceptable program otherwise it will print out helpful error messages.

The J Language

The J language is a stack-oriented language loosely inspired by Forth. You may want to look at the following Wikipedia article for a quick description of stack oriented languages.

http://en.wikipedia.org/wiki/Stack-oriented_programming_language

In a stack oriented language all operands are passed on the stack and most operations end up manipulating the stack. Here are a few quick examples of J program fragments.

Egs. 1:

```
7 5 4 + *
```

This program pushes the values 7, 5 and 4 onto the stack (stack state: 7 5 4 – note that the rightmost element in this list, 4, is viewed as being at the top of the stack) then adds the top two elements (stack state: 7 9) then multiplies the results (stack state: 63). This type of operational syntax is sometimes referred to as Reverse Polish Notation (RPN) and you see it used on many revered HP calculators.

Egs 2:

```
5 7 gt if 12 else 25 endif        ; this is a comment
```

This program pushes 5 and 7 on the stack (stack state: 5 7) then compares the top two elements to determine if the topmost element is greater than the next element (stack state: true/1). The if statement then queries the top element of the stack and if it is non-zero, which it is in this case, it pushes the value 12 on the stack otherwise it would push the value 25. So at the end of this program the stack state would be (stack state: 12). Note that the true value is consumed by the if operator. Note also the comment at the end of the line – everything from the semi-colon to the end of the line should be disregarded by the compiler.

Parsing J source files:

As you can see from the examples a J source file can be thought of as a sequence of ***tokens*** (egs. 7, -9, if, eq, +, defun) separated by whitespace characters (spaces, tabs, newlines). Tokens can denote literal values or operations. You will want to proceed by writing a program that attempts to read in the tokens from the file one at a time and generates the appropriate assembly code in response to those tokens. For the sake of simplicity, you can assume that no token will contain more than 200 characters. Remember that you do need to handle comments that follow semi-colons. Note that blank lines, tabs and spaces can be used to format the J code so that it is more readable. You may want to write a specific function for reading the next token from the source file. You may find the routines in <ctype.h> useful.

Elements of the J language:

Operands:

For simplicity, all of the operands in the J language are 16 bit words. There are no other data types.

Literals:

Tokens consisting entirely of digits optionally preceded by a minus sign are interpreted as decimal numbers and should be pushed on the stack in the order they are encountered. Additionally, you should implement hexadecimal literals which begin with the string ‘0x’ followed by a sequence of hexadecimal characters. You should read these hex values and then push them on the stack just as you would a decimal literal.

Arithmetic Operations:

The J language has tokens that denote the basic arithmetic operations provided by the LC4 assembly language.

- + : ADD
- - : SUB
- * : MUL
- / : DIV
- % : MOD

In each case the value at the top of the stack is used as the first operand, Rs, and the next item on the stack is used as the second operand, Rt. These two values are popped from the stack and the value of the operation is pushed onto the stack.

Comparisons:

Let a denote the element at the top of the stack and b denote the next element. There are 5 comparison operators in the J language and they are listed below.

- lt : a < b
- le : a <= b
- eq : a == b
- ge : a >= b
- gt : a > b

Note that the top two values are popped off the stack and replaced by the result of the comparison which is either a 1 if the comparison produces a true value or 0 if it is false. <u>Note that the values a and b are viewed as 2C values for the purposes of these comparisons</u>.

Logical Operations:

The J language supports the 3 basic binary Boolean operations through the tokens **and**, **or** and **not**. The and and the or operations take the two top values on the stack, compute their respective operations in a bitwise manner using the corresponding LC4 instructions and then place the result back on the top of the stack. The not operation does much the same thing but it only consumes the top element of the stack.

Stack Operations:

Stack oriented languages always have operations to manipulate the state of the stack. The J language provides many of the basic stack operations offered by Forth and these are listed below.

- **drop** : drops the top element of the stack
    - egs:
        - stack state before drop: 7 -6 3
        - stack state after drop : 7 -6
- **dup** : Duplicates the top element of the stack
    - egs
        - stack state before dup: 7 -6 3
        - stack state after dup : 7 -6 3 3
- **swap** : swaps the top two entries of the stack
    - egs:
        - stack state before swap: 7 -6 3
        - stack state after swap : 7 3 -6
- **rot** : rotates the top 3 elements of the stack
    - egs:
        - stack state before rot: 7 -6 3
        - stack state after rot : -6 3 7

- **argN** : This command copies an entry from one of the argument slots of the function to the top of the stack. Egs arg1 copies the first argument, the one right below the RV slot to the top of the stack arg2 copies the second argument to the top of the stack, arg3 would copy the third argument etc. The value N can range from 1 to 20 ie arg1, arg2, arg3, …, arg19, arg20 are all legal commands.

If statements:

The J language supports if .. else .. endif constructs as mentioned earlier. As in other languages the else clause is optional. The syntax is as follows;

if < block A > else < block B> endif

When the if statement is executed the system consumes the top element of the stack if this value is non-zero, ie true, then the statements in block A are executed, otherwise the statements in block B are executed. You may want to look at how lcc compiles if statements into LC4 assembly. You will notice that this is done by branching to appropriate unique labels. Note that a given J procedure or program may have many if statements so you must be sure to generate unique labels in the assembly code for each one. One way to keep track of what is going on is to keep a count of the number of if, and endif tokens that you have encountered as you parse the file from start to finish. If you think about this carefully you will be able to correctly deal with multiple if statements as well as nested if statements. You will probably want to use the file name or procedure name in the labels to make sure that the labels are unique even when the code is spread over many files.

Comments:

As we mentioned earlier we will use comments in the J code to document the design. If you encounter a semi-colon in the file all content between that point and the end of the line is regarded as a comment and is ignored by the compiler.

Egs;

```
1 2 3 4 + -  ; This is a comment -  everything up to the end of the line
```

Functions:

Like C, J is actually intended to operate as a procedural language which means that all of the code needs to be packaged in functions which will be turned into blocks of code just the way lcc does. Function definitions are a bit simpler in J since we can assume that the calling context has already pushed any required arguments onto the stack before calling the function, hence there is no need for an argument list.

Function definitions are started with the **defun** token as illustrated in the following example:

```
defun square ; a function to compute the square of the value at the top of the stack
arg1 ; fetch the input argument, n from below the FP, RA and RV slots
dup * ; stack state  : n*n
return ; copy n*n into RV slot and return
```

Once the function has been defined we can invoke it by its name egs;

```
4 square  ; computes the square of 4 – stack state after call: 4 16
```

Function names must start with an alphabetical character and can consist of alphabetical characters, underscores '_' and digits only. These identifiers are case sensitive.

The **return** token is used to copy the value at the top of the stack into the RV slot associated with the current frame. It also invokes the function epilogue to reset the stack and frame pointer and to return from the function following the lcc convention. Note that every function should terminate with a return at some point.

Functions are invoked by simply typing their name as shown above. Once again you are expected to use exactly the same calling convention that the lcc compiler does where R5 acts as the frame pointer and R6 acts as the stack pointer. The only small difference is that when you generate code to call a subroutine you should ensure that once the function has returned the return value of that function is included at the top of the stack instead of being just beyond it as is the case with lcc. This is easily accomplished by modifying the stack pointer after the subroutine has returned. Remember that it is the calling functions job to clean up arguments and return values as it sees fit once the subroutine has returned. In the J language we will need to handle this explicitly in the code – that is the jc compiler should not automatically generate code to pop the return value or the function arguments.

Note that since we are adopting the lcc calling convention we should be able to call functions written in C and compiled with the lcc compiler from within our J programs and vice versa. This will be a useful way to add functionality to our system.

Getting Started:

In order to give you a couple of ideas about how to get started in thinking about and implementing this program we have provided you with a header file called token.h. This file defines a type called token which is intended to represent the kinds of tokens you will encounter in a J program. It also declares a function called read_token which reads the next available token from the file. It should return a 0 if it was successful and a non-zero value if it encountered an error, you can imagine returning different values for different kinds of errors. The way the compiler proceeds is that it repeatedly reads tokens from the file and for each token it generates the requisite assembly. You can look at what the LCC compiler does with your C files for inspiration. Here is the pseudo code for the compiler – not how short it is, at least conceptually.

```
Pseudo code for J compiler
While read_token returns 0
        Consider the token you have just read and emit corresponding assembly into
        the output ASM file. (To do this consider each type of token and ask what
        assembly you need to emit to translate that element)
```

Error Handling

Your program should be able to handle any well formed J program so that's what you should focus on. We will not be doing a lot of testing with malformed J programs so we are not expecting extensive error handling to catch every possible problem. That being said you probably want to do some basic error handling for your own sanity like checking for illegal tokens that you can't process. Similarly, you may want to check for missing return statements and if statements that aren't properly delimited with endifs.

Testing:

As usual you can't really tell if your code works if you don't test it. To this end we are providing you with a variety of J programs of varying levels of complexity. We also provide you with an os.asm a libc.asm that contain the operating system and a set of I/O functions you can call and printnum.c that contains a C function for printing function that will be called by many of the sample J programs. We also provide script files that can be used in PennSim to run the code. (see the associated zip file on Canvas)

You would be well advised to write your own test cases as well to test your code since we will be testing on more than just the examples that are handed out.

Your Code.

Once again we expect you to use at least 3 source files for this assignment and you must provide a Makefile that produces the final executable, jc, that is if we type make jc in your source directory it should build the executable jc from scratch. You can certainly split your code over more than 3 source files if you prefer. In addition you should have a target called clean such that make clean removes the trace program if it exists along with any other compiler outputs. Here is an example of what the requisite lines in your makefile should be

```
clean :
	rm jc *.o
```

Extra Credit Challenge {20 pts}

You can score up to 20 extra credit points by writing a program that, when run on PennSim produces a graphical animation of a solution to the Towers of Hanoi problem using at least 5 discs. i.e. When your program runs it should produce a nice animation on the PennSim graphics display of discs moving about in an orderly manner to solve this problem. The constraints are as follows:
1) You must use a recursive algorithm to solve this - (this is the easiest way anyway)
2) Your program must be a combination of C and J. You will have to write C routines to do things like drawing the discs but the part of the code that decides how to move the discs must be in J. That is you can't simply write the whole thing in C and then call the solver from J with a single function call.

On the bright side the libc and os code that was provided contains implementations of useful functions like lc4_get_event, lc4_draw_rect and lc4_reset_vmem.

Submission instructions.

When you're done with your code, please submit the assignment on Gradescope. Both your executable and your make target MUST be named "jc". Submit it by uploading each individual file (not in a folder) to the Gradescope submission portal.