**Computer Science 52**

**Assignment 4**

*Due Friday, October 7, 2017, at 5:00 PM*

This is an assignment about data representation and computer architecture. It uses the CS52 Machine to demonstrate how programs are run on a real computer and how recursion is implemented.

**Reading**

- *Bits and Logic*, linked from the course Resources page. Skip the part on floating point numbers for now.
- *The CS52 Machine*, also linked from course Resources page. Sections 1 and 2, and Appendices B and C are the most relevant.
- The sample programs linked on the course Resources pages.
- The appendix to this assignment, in which Professor Kauchak shares insightful suggestions on how to write CS52 Machine programs.

**Preparation**  The CS52 Machine simulator is a Java application that should run on most platforms. It, along with the documentation and sample programs, is linked from the course webpage page in the resources section. That material is also on the lab computers in the directory `/common/cs/cs052/cs52-machine`.

There is no template or check file for this assignment.

Aquamacs note: Often Aquamacs uses a variable-width font, making it difficult to line up columns. To force it to use a fixed-width font, follow the menu selections Options → Appearance → Auto Faces and uncheck Auto Faces.

Write all of your programs to run on the CS52 Machine with the default memory size of 512 bytes.

**Submission**  Write and submit (in the usual way) five clear and correct files, nonuple.a52, power3.a52, oddfact.a52, ackermann.a52, and asgt04-5.txt. Submit them as individual files; *do not zip*! The first four should be well-annotated; there is no need for documentation in the last.

![XKCD comic](https://xkcd.com/244)
1. [3 points] Write a CS52 Machine program `nonuple.a52` that takes a single value as an input and returns that value multiplied by 9. We want you to have practice writing subprograms, so your program must have

- a function `triple` that that returns its argument tripled and
- a main section that reads the input, calls `triple` twice, and writes the result.

The function `triple` must obey the register-use and stack conventions described in class and the CS52 Machine documentation. Have your function `triple` compute its result by performing two additions. Do not use a multiplication routine.

CS52 wants a value > -7  
CS52 says > -63

2. [3 points] Write a CS52 Machine program `power3.a52` that takes as input a number \( n \) and prints out \( 3^n \). To accomplish the task write a function that computes powers of 3 by following the recursive pattern suggested by the SML function below.

```sml
fun power3 k = if k < 0
    then 0
    else if k = 0
    then 1
    else triple(power3(k-1));
```

“Tripling” may be implemented by adding a number to itself. You may borrow the function `triple` from the previous problem or just write a few lines of code. Do not use multiplication.

Your program must call a recursive function that accurately reflects the pattern above. Here “recursive” means that the function saves the return address on the stack and eventually jumps back to it, and the body of the function contains a call to itself.

For example,

CS52 wants a value > 5  
CS52 says > 243

3. [5 points] Write a CS52 Machine program `oddfact.a52` that computes the “odd factorial function,” defined as follows.

\[
f(k) = \begin{cases} 
0 & \text{when } k < 0, \\
1 & \text{when } k = 0, \\
k \cdot f(k - 1) & \text{when } 0 < k \text{ and } k \text{ is odd, and} \\
f(k - 1) & \text{otherwise}
\end{cases}
\]
For example,
\[
\begin{align*}
  f(6) &= \ f(5) \\
        &= \ 5 \cdot f(4) \\
        &= \ 5 \cdot f(3) \\
        &= \ 5 \cdot 3 \cdot f(2) \\
        &= \ 5 \cdot 3 \cdot f(1) \\
        &= \ 5 \cdot 3 \cdot 1 \\
        &= \ 15
\end{align*}
\]

As in Problem 2, your program must call a recursive function that reflects the pattern above.

Fashion your program after the sample programs, and adhere to the conventions for register use. In particular, see the sample factorial program for an example of using the library mllib.a52 for multiplication. You can determine whether an integer is odd by computing its bitwise-and with 1.

CS52 wants a value > 6
CS52 says > 15

4. [7 points] The Ackermann function is defined for natural numbers \( m \) and \( n \) as follows:

\[
A(m, n) = \begin{cases} 
0 & \text{if } m < 0 \text{ or } n < 0, \\
  n + 1 & \text{if } m = 0 \text{ and } 0 \leq n, \\
  A(m - 1, 1) & \text{if } 0 < m \text{ and } n = 0, \text{ and} \\
  A(m - 1, A(m, n - 1)) & \text{otherwise}
\end{cases}
\]

Write another CS52 Machine program that takes two values, first \( m \) and then \( n \), and prints out the corresponding value of the Ackermann function. Place your work in a file named \texttt{ackermann.a52}.

CS52 wants a value > -3
CS52 wants a value > 1
CS52 says > 0

CS52 wants a value > 3
CS52 wants a value > 1
CS52 says > 13

The function grows very quickly, and your program will make \emph{lots} of recursive calls. The table below shows some values of the function. The value \( A(4, 2) \) requires nearly 20,000 decimal digits; you can see the actual value in a document linked from the CS52 Machine section of the course Resources page.
Your program will actually produce results for only the smallest values of \( m \) and \( n \). If you squeeze in as much stack space as you can into the default memory size of 512 bytes, you ought to be able to compute \( A(3, 3) \) and \( A(4, 0) \) but probably not \( A(3, 4) \) or \( A(4, 1) \). Nevertheless, your code must be structured as an accurate representation of the recursive definition above.

Contextual note: The function given here, more properly called the Ackermann-Péter function, is an important example in the theory of computability. It is computable and gives a result for all pairs of natural numbers, but it cannot be defined using only simple one-variable recursion. In the language of computability theory, it is a total recursive function but not a primitive recursive function.

5. [5 points] This exercise gives us a glimpse into unsafe languages. Study the program asgt04-5.a52, both shown below and linked from the course webpage. The heart of the program is a function \texttt{accumulate} which takes values from the user and stores them in an array of size four. When the user provides the value zero, the function returns 47.

There is a block of code at the label \texttt{nevercalled} that presumably will never be invoked. Your task is to find a sequence of input values that force the program to execute those instructions and print the value −47. Do not change the program.

Format your solution in a file named \texttt{asgt04-5.txt} with one integer (in decimal) per line. Include only the integers—including the final zero—without any comments or other text.

Hint: Think about where things are on the stack. There is nothing to prevent you from providing more than four non-zero integers.

ackermann: a bust
for sml once more i lust!
in recurse we trust
Sam Rubin ’19, CS52 poet laureate
This short program illustrates how programs can be made to misbehave. The main function simply calls accumulate() and prints the value returned.

The function accumulate allocates an array of four integers on the stack and fills it with values from the user. To keep things simple, accumulate does nothing with those values and returns the constant 47. (You can imagine in a more realistic program that it would return something like the sum of the values in the array.)

There is a short sequence of instructions at the label nevercalled which, under circumstances that a programmer might consider normal, will never be executed. One of the tasks in this exercise is to force those instructions to be used by jumping to the location nevercalled.

```c
void main() {
    int result = accumulate();
    write(result);
}

int accumulate() {
    int a[4];
    int j = 0;
    int n = read();
    while (n != 0) {
        a[j] = n;
        j++;
        n = read();
    }
    return 47;
}
```

```asm
lcw r1 stack ; set up the stack
lcw r2 accumulate ; call the function accumulate
cal r2 r2 ;
sto r3 r0 ; write the result
hlt ; quit

nevercalled
neg r3 r3 ; change the result
sub r2 r2 4 ; manufacture a return address
    ; (assumes a jmp to nevercalled, so that the location nevercalled is still in r2)
jmp r2 ; return

accumulate
psr r2 ; save the return address
sub r1 r1 8 ; make space for an array of four integers
add r2 r1 2 ; r2 is an index into the array
brs acctest ; go for first test

acclloop
sto r3 r2 ; store it in the array
add r2 r2 2 ; increment the array index

acctest
loa r3 r0 ; get another value
bne r3 r0 acclloop ; if nonzero, go back for more
add r3 r0 47 ; return the value 47
add r1 r1 8 ; recover the original stack pointer
pop r2 ; recover the return address
jmp r2 ; return to caller
dat 16 ; no need for a huge stack

stack
```
Hints on Debugging the CS52 Machine Programs

Eventually (hopefully already!) you’ll get the hang of the CS52 Machine’s assembly language, the stack, and how to make function calls. Once you reach this stage, if all goes well, you’ll code up your functions and with minimal tweaking, and they’ll work!

While I try to remain optimistic, most of you will still probably run into one last bug or two (I almost always do!) that you can’t figure out by just looking at the code. If you reach this point and you really understand what’s going on in your code, but can’t find the bug, then the next step is to step through the execution of your code and follow the execution. If you do this a step at a time, you’ll eventually find the problem.

A few thoughts on this:

- Open your code in the simulator and run it a line at a time using the single-step button.
- Get out a piece of paper and write down a place for \( r_2 \), \( r_3 \) and the stack. As you execute each line of code, update what you think should be happening on the piece of paper and then double check that’s what happened in the simulator. If they’re different, figure out why!
- Remember, \( r_1 \) holds the location of the next location where a value would be put on stack. To view the stack:
  1. Look at the memory location stored in \( r_1 \), i.e. the value in \( r_1 \).
  2. Look at the data view on the left of the simulator and find the address (left part before the colon) corresponding to the address in \( r_1 \).
  3. The address in \( r_1 \) is **not** part of the stack. The first value in the stack is the address immediately below the address in \( r_1 \) (e.g. if \( r_1 \) is 00da then the first value of the stack is 00dc). Remember, the stack will grow towards smaller memory addresses, so the stack itself will be in the larger memory addresses.

Using these things, you should be able to step through your code a line at a time. If you understand what’s supposed to be happening, then this can be very, very helpful at identifying small bugs.

My personal experience: When I first wrote Ackerman, I had a small bug in my function (I had written “\texttt{loa r2 r1 6}” when it should have been “\texttt{loa r2 r1 4}”) and I only found it by doing what I describe above. I know it may seem a bit tedious, but I promise you will find the bug much, much faster using this approach than most others.