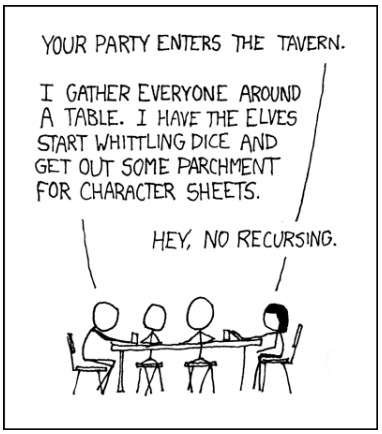
Computer Science 54

Assignment 5

Sunday, October 9 @ 11:59pm



https://xkcd.com/244/

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.

Partners On this assignment you may (and I'd encourage you to) work with a partner. If you choose to, you must both be there anytime you're working on the assignment.

Write and submit (in the usual way) **four** files, `nonuple.a52`, `power2.a52`, `oddfact.a52`, and `asgt04-4.txt`. Submit them as individual files; *do not zip them up!*. The first three should be well-commented; there is no need for documentation in the last.

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. Please try and download it ASAP to make sure you can get it running. If you don't have Java installed on your computer, you're need to install a Java JRE (run-time envivonment).

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

A note on debugging in the CS52 machine

Eventually (hopefully already!) you'll get the hang of the CS52 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, they'll work!

While I try and 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 "play" (arrow) button.
- Get out a piece of paper and write down a place for `r2`, `r3` 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, `r1` 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 `r1`, i.e. the value in `r1`.
  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 `r1`.
  3) The address in `r1` is **not** part of the stack. The first value in the stack is the address immediately below the address in `r1` (e.g. if `r1` 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.

"When I first wrote one of the more complicated functions, I had a small bug in my function (I was doing a "`loa r2 r1 6`" when it should have been"`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."
–Dr. Dave

Course feedback

We're about a third of the way through the course and I wanted to checkin and see how things are going. If you want (it's anonymous), take 5 minutes and let me know how things are going:

`https://forms.gle/4RPWHcomZ4bcdsiQA`

The fun stuff

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 program `power2.a52` that takes as input a number n and prints out 2^n. To accomplish this write a function that computes powers of 2 by following the recursive pattern suggested by the SML function below.

```
fun power2 k =
     if k < 0 then
          0
     else if k = 0 then
          1
     else
          double(power2(k-1));
```

"Doubling" may be implemented by adding a number to itself; there need not be a separate doubling function nor should you 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 > 32
```

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,

$$\begin{aligned} f(6) &= f(5) \\ &= 5 * f(4) \\ &= 5 * f(3) \\ &= 5 * 3 * f(2) \\ &= 5 * 3 * f(1) \\ &= 5 * 3 * 1 \\ &= 15 \end{aligned}$$

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.

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 `mullib.a52` for multiplication. You can determine whether an integer is odd by computing the bitwise-`and` with 1.

```
CS52 wants a value > 6
CS52 says > 15
```

4. [5 points] This exercise gives us a glimpse into unsafe languages. Study the program `assign5-4.a52`, both shown below and linked from the course webpage. The heart of the program is a function `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 `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.

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

Format your solution in a file named `asgt04-4.txt` with one integer (in

decimal) per line. Include only the integers—including the final zero—
without any comments or other text.

```
;
; assign05-4.a52
;
; CS Profs
;
; 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.
;
;
;    void main() {
;        int result = accumulate();
;        write(result);
;    }
;
;    int accumulate() {
;        int a[4];
;        int j = -1;
;        int n = read();
;        while (n != 0) {
;            j++;
;            a[j] = n;
;            n = read();
;        }
;        return 47;
;    }
;
        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
        psh r2              ; save the return address
        sub r1 r1 8         ; make space for an array of four integers
        mov r2 r1           ; r2 is an index into the array
        loa r3 r0           ; get a value
        beq r3 r0 accdone   ; quit if the value is zero
accloop
        add r2 r2 2         ; increment the array index
        sto r3 r2           ; store it in the array
        loa r3 r0           ; get another value
        bne r3 r0 accloop   ; if nonzero, go back for more
accdone
        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
```