Adam Stammer

**Abstract:** Subroutines allow a program to save space and time, and frequently simplify code making it easier to understand. Recursion takes it a step further. We used a predefined recursive exponentiation algorithm to computer one input raised to the other input.

**Title:** Recursive Exponentiation

**Purpose:** To practice and display understanding of subroutines in a recursive way, utilizing the stack, a predefined stack frame, and some of the more complex operands we have available on the PDP-11.

**Inputs:** One positive 2’s complement integer (N as described in the pseudocode)

One positive 2’s complement integer (M as described in the pseudocode)

**Outputs:** One positive 2’s complement integer (M^N)

**Pseudocode:**

Q = REXP(M, N)

RETURN Q

REXP(M, N)

IF N==0

RETURN 1

ELSE IF N%2 == 0

C=REXP(M, N/2)

D = C

FOR I = C; I < 0; I++

D = D\*C

RETURN D

ELSE

C = REXP(M, N-1)

C = M\*C

RETURN C

**Discussion of Results and Lessons Learned:** I’ve spent quite a bit of time debugging various programs but very few have been as tedious and annoying as this one. Some of my mistakes were simple typos, others a misinterpretation of the pseudo code, and others yet were problems I thought lessons learned long ago. For quite some time I was using the wrong register of multiplication output, something I should’ve known to get right the first time. My most time consuming of blunders was a classic forgetting that the pdp11 uses octal, not decimal, thus making any stack index past 6 not work as expected. Now it might not be pretty, or have any error checking, but it mostly works.

I did fiddle around a bit with the differences between pass-by-value and pass-by-reference arguments, and I found this to be very easy to mix up. So much of the work done in subroutines has to explicitly be done before, in, and after the function that it’s easy to forget something simple. It only takes one missing “@” to mess up an entire program.

I am still fascinated by the use of a stack in our running subroutine/argument hierarchy. Layering like this has a lot functionality but it can be difficult getting an assembler program big enough to demonstrate this without the program getting too complex. I think our recursive example is rather straight forward though. The stack builds, one stack frame on top of the last, until finally the last piece, in our case when n = 0), and the entire tower of stack frames just falls down to build our answer. It puts both a visual and a logical understanding to recursion as a whole. And I was even able to see that, in memory, here. It was slightly tedious examining all of the words, but it was a vital tool in my debugging, and rather cool to see in action.

Since this program doesn’t catch any overflows at all, or handle negative input, the test cases weren’t that exciting. After testing multiple small inputs, and getting the expected the results, I then tried to push past overflow. I did this under cardinal output interpretation, but one could consider the overflow point to be 32768 rather than 65536 if limited to 2’s comp. The sqrt of 65536 is 256, so I tried M=256 and N=2 expecting the bare minimum for an overflow, and that’s exactly what happened. The same can be said for M=2, N=16. For both of these I also showed slightly under this overflow by decrementing M and then N respectively. This overflow remains uncaught and would certainly not be acceptable in a release program, but for our purposes I’m satisfied with this program.

I went back and added a test case that assumes 2^15 is overflow (strict 2’s comp), which behaved as expected. I also added some negative test cases just to see that they do indeed fail.

Attached is the generated list file of my program and a copy of the log file running through the majority of my test cases showing both input and output.