Adam Stammer

**Abstract:** Prime numbers serve multiple purposes but one of the most common applications leverages the difficulty in computing large prime numbers. With our limited memory and limited instruction set it is impractical to the PDP11 to generate these prime numbers but it can server as a useful tool in teaching new concepts and practicing what we already know.

**Title:** Prime Number Generator (Program Four)

**Purpose:** To further our understanding of the instruction set we have available on the PDP11. To get experience converting higher level language code to lower level language code. To practice nested loops in lower level coding. To gain examples of the difficulty in computing prime numbers.

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

**Outputs:** One positive 2’s complement integer (the Nth prime number or -32768 as error code)

**Pseudocode:**

See the attached java code (on the next page). While this java code doesn’t translate directly to the assembler program in question, it is what I worked off of.

**Discussion of Results and Lessons Learned:** It was almost painful being so restricted with our prime number generation algorithm. This on is arguably the easiest to understand but even with our relatively low numbers and our simulator running on a modern computer, I was able to feel how slow this program really is. After around the 1000th prime number, I would have to wait for the program to finish before I could examine the output, which became more and more annoying the higher up in primes I went. Thankfully, after the overflow case is hit, it stops taking longer to overflow and levels out in duration. This was a nice example of how hard it can be to compute primes though, if on a smaller scale than modern applications.

My test cases started with negatives, especially the extremes to make sure they crashed with the proper error code. This wasn’t exhaustive but good enough for me. Then zero, since this isn’t negative but still should produce the error code. Again, I got what I expected. I was far less focused on loop optimization this time around since I was more focused on the program working. That said, I don’t think these loops are horribly inefficient.

Then I sought out the overflow fence. When does the answer become too big to store? Using known tools, I found that the 3512th prime number is slightly under our 2’s comp. MAX\_INT, and that the 3513rd prime is slightly over our 2’s comp. MAX\_INT. This fence was an obvious test choice, and ran as expected, but we’ve run into the same problem as I had on the last few assignments: using preexisting solutions to find the limits of a known problem. I could’ve directly found the limit for our algorithm, or used the age old guess and check method, but it was much easier to find an existing list of prime numbers. With this particular example though it kind of defeats much of the value in prime numbers. If we already know what they are, they can no longer be that useful in security. But my overflow error cases were caught, including close to MAX\_INT, so in practice this fence could be found though general program use.

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.