Where's the center of the solar system??!

Changes and Adjustments from Phase 2

I wrote the `generateRandomNumbers` function to generate larger arrays for x and y so that I could properly test my program. The x and y arrays are now 2^28 numbers in length. I will assume that all arrays inputted into my `meanArray` function are divisible by 16.

Benchmarking the original program

I am using my personal desktop, which contains an AMD Ryzen 7 5800x (team red!) with no overclocking or underclocking.

I am benchmarking the two `meanArray` function calls (and a few unrelated instructions that should not affect the timing in any significant way). On the unoptimized phase 3 code, I get the following results after running my program three times.

175ms **177ms** **175ms** Average: **176ms**

```
    ; Find x_mean
    mov rcx, x
    mov rdx, 268435456
    call meanArray
    mov x_mean, rax

    ; Find y_mean
    mov rcx, y
    mov rdx, 268435456
    call meanArray
    mov y_mean, rax  ≤ 175ms elapsed
```

Optimization #1 – Loop unrolling and cheaper instructions

I stripped the loop down so that it used as few instructions as possible (removing a cmp and mov instruction) and unrolled it so that it adds 4 numbers in 1 iteration. I got the following results after running my program three times.

177ms **173ms** **173ms** Average: **174ms** <u>Speed increase: ~**1%**</u>

```
    ; rax contains the running total of the array
    ; rbx is my loop counter
    mov rax, 0
    mov rbx, rdx
    dec rbx

meanArrayLoop:
    mov r12, rbx
    mov r13, rbx
    mov r14, rbx

    dec r12
    sub r13, 2
    sub r14, 3

    add rax, [rcx + rbx * 8]
    add rax, [rcx + r12 * 8]
    add rax, [rcx + r13 * 8]
    add rax, [rcx + r14 * 8]

    sub rbx, 4
    jns meanArrayLoop
```

Optimization #2 – SSE/AVX parallelization

I tried using parallel additions to speed up my loop. I now have two running totals, both of which are stored together in xmm0, each of them adding up exactly half of the array elements. I combine the totals at the end of the loop into RAX on lines 66-68. This optimization alone did not really increase the speed of the program, but it was a useful steppingstone for optimization 3.

177ms **174ms** **174ms** Average: **175ms** <u>Speed increase: ~**1%**</u>

```
48      ; Clear xmm0
49      mov [rsp], rax
50      mov [rsp + 8], rax
51      movdqu xmm0, [rsp]
52
53  meanArrayLoop:
54      ; Load the next 2 64-bit numbers into an xmm registers
55      movdqu xmm1, [rcx + rbx * 8]
56
57      ; Add the two numbers to our running total - we're performing two additions
58      ;   in parallel here.
59      paddq xmm0, xmm1
60
61      add rbx, 2
62      cmp rbx, rdx
63      jnz meanArrayLoop
64
65      ; Take the two running totals and add them together to get the final result
66      movdqu [rsp], xmm0
67      add rax, [rsp]
68      add rax, [rsp + 8]
69
70      ; Division in x64 assembly is a little weird - we're dividing a 128 bit
```

Optimization #3 – SSE/AVX parallelization and loop unrolling

I took the code from optimization #2 and unrolled it so that I could perform 8 iterations of the loop at once, allowing for a total of 16 additions per iteration of the loop. Surprisingly, this had some really good results.

146ms **146ms** **148ms** Average: **147ms** Speed increase: **~20%**

```

meanArrayLoop:
    ; Load the next 16 64-bit numbers into some xmm registers
    movdqu xmm1, [rcx + rbx * 8]
    add rbx, 2
    movdqu xmm2, [rcx + rbx * 8]
    add rbx, 2
    movdqu xmm3, [rcx + rbx * 8]
    add rbx, 2
    movdqu xmm4, [rcx + rbx * 8]
    add rbx, 2

    movdqu xmm5, [rcx + rbx * 8]
    add rbx, 2
    movdqu xmm6, [rcx + rbx * 8]
    add rbx, 2
    movdqu xmm7, [rcx + rbx * 8]
    add rbx, 2
    movdqu xmm8, [rcx + rbx * 8]
    add rbx, 2

    paddq xmm0, xmm1
    paddq xmm0, xmm2
    paddq xmm0, xmm3
    paddq xmm0, xmm4
    paddq xmm0, xmm5
    paddq xmm0, xmm6
    paddq xmm0, xmm7
    paddq xmm0, xmm8

    cmp rbx, rdx
```