//Charles Buyas, cjb8qf, 4-12-17, inlab9.pdf

I chose to begin with analyzing the optimizations. I wrote a short program that utilized the prelab algorithm, and solved it iteratively.

```
// Charles Buyas, cjb8qf, 4-11-17, inlab.cpp
#include <iostream>

using namespace std;

int main() {
  int x;
  int counter;
  counter = 0;
  cin >> x;
  while (x > 1) {
    if (x%2 != 0) {
      counter++;
      x = (3*x) + 1;
    }
    else {
      counter++;
      x = x/2;
    }
  }
  return 0;

}
```

When I compiled it with assembly, I created two separate s files to compare the unoptimized and optimized assembly side by side.

The first difference I noticed was the way the assembly handled the “x = (3*x) + 1”. The unop and op code was very different:

```
# BB#3:                                 #   in Loop: Header=BB1_1 Depth=1
        mov     eax, dword ptr [rbp - 12]
        add     eax, 1
        mov     dword ptr [rbp - 12], eax
        imul    eax, dword ptr [rbp - 8], 3
        add     eax, 1
        mov     dword ptr [rbp - 8], eax
        jmp     .LBB1_5
```

Here (above) we see the unoptimized version of the first if statement within the while loop that does the required math if the number is odd. We see that the code uses the “imul” and “add” commands in the unop version of the code.

```
.LBB0_2:
        lea     eax, [rax + 2*rax + 1]
```

Here (above) we the optimized version of the math operation uses “lea” instead, allowing the register to do both operations of multiplying and adding all in one line of code. This is optimized because the operation can be performed faster and requires fewer lines.

Basically the computer knows that multiplying by 2 is the easiest thing it can do. If you multiply by 2, all the computer has to do is shift over the binary bits by 1. So, in order to speed up the operation of multiplying by 3, it uses the easy 2 multiplication and just adds on another rax. Since multiplication is just adding over and over again, this saves time with the 2 multiplication requiring only bits to be shifted. Therefore this form of computation is faster and optimized.

The next difference is how the assembly code handles the if statement that determines if x was odd: “if (x%2 != 0)”.

```
.LBB1_1:
        cmp     dword ptr [rbp - 8], 1
        jle     .LBB1_6
```

Here (above) we see the unoptimized version of assembly. It compares the data value x, which is stored at [rbp - 8], and compares it to 1.

```
.LBB0_1:

        test    al, 1
        jne     .LBB0_2
```

Here (above) we see the optimized version of assembly. This uses the “test” function to simply check the zeroth digit of binary and see if a ‘1’ is active or a ‘0’ is active. If it’s odd, then it jumps to the operation we can see in the optimized version of the “x = (3*x) + 1” above.

One optimization I found quite odd was the use of leaving space to avoid buffer overflow. In the unoptimized version we see the expected space being placed:

```
.Ltmp5:
        .cfi_def_cfa_register rbp
        sub     rsp, 32
        movabs  rdi, _ZSt3cin
        lea     rsi, [rbp - 8]
        mov     dword ptr [rbp - 4], 0
        mov     dword ptr [rbp - 12], 0
        call    _ZNSirsERi
        mov     qword ptr [rbp - 24], rax # 8-byte Spill
```

It saves space for 8 bytes of overflow. In the optimized version, however, we don’t see any space set aside for overflow, probably because it has the program in its entirety and knows that there won’t be overflow:

```
.Ltmp0:
        .cfi_def_cfa_offset 16
        lea     rsi, [rsp + 4]
        mov     edi, _ZSt3cin
        call    _ZNSirsERi
        mov     eax, dword ptr [rsp + 4]
        cmp     eax, 2
        jl      .LBB0_6
        .align  16, 0x90
```

One final thing I took notice of was the use of the shift bits operators in the optimized version, “sar” and “shr”:
Below is the unoptimized version of code that simply divides by two using the “div” operation on the memory location of x and the number 2 that was stored earlier in the function by using eax and the stack.

```
# BB#2:                                 #   in Loop: Header=
        mov     eax, 2
        mov     ecx, dword ptr [rbp - 8]
        mov     dword ptr [rbp - 28], eax # 4-byte Spill
        mov     eax, ecx
        cdq
        mov     ecx, dword ptr [rbp - 28] # 4-byte Reload
        idiv    ecx
        cmp     edx, 0
        je      .LBB1_4
```

The optimized version, however, uses “shr” and “sar” to shift the bits in a clever way.

```
# BB#3:
        mov     ecx, eax
        shr     ecx, 31
        add     ecx, eax
        sar     ecx
        mov     eax, ecx
        jmp     .LBB0_4
        .align  16, 0x90
```

The “shr” shifts the bits over, and the “sar” is used to ensure that the signed bit of the number remains the same throughout the shift. It then jumps back up to the top of the loop in order to continue through the function.