//Charles Buyas, cjb8qf, 4-13-17, postlab9.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.

I then wrote a function that would allow me to analyze Dynamic Dispatch in assembly. I compiled using “clang++ -S -m64 -mllvm --x86-asm-syntax=intel postlab.cpp” and began looking through the resultant assembly file.

The original c++ file is quite simple:

```
1  // Charles Buyas, cjb8qf, 4-13-17, postlab.cpp
2
3  #include <iostream>
4  #include <cstdlib>
5
6  using namespace std;
7
8  class A {
9  public:
10   virtual void car() {
11   }
12   virtual int speed() {
13     return 1;
14   }
15 };
16
17 class B : public A {
18 public:
19   virtual void car() {
20   }
21   virtual int speed() {
22     return 2;
23   }
24 };
25
```

```
25
26 int main () {
27     int which;
28     A *bar;
29     if ( which ) {
30         bar = new A();
31     }
32     else {
33         bar = new B();
34     }
35     bar->car();
36     bar->speed();
37     return 0;
38 }
```

All this does is create two separate classes A and B, respectively. And in each class there are the two methods named “car” and “speed”. The “car” method is a void method that does and returns nothing, while the “speed” method is of type ‘int’ and returns either a 1 or a 2 depending on which class it comes from.

Basically the inheritance that occurs through dynamic dispatch, in this case, works like a parent and child. Parent class A has certain attributes defined within it, and so does child class B. While child class B can inherit things from its parent (for example I inherited my blue eyes from my father and mother), parent class A does not inherit anything from child class B (my father didn’t get his blue eyes from me).

In one test case, I commented out the line “bar->speed();” in order to analyze how it handled the call of “car”. In another I commented out the line “bar->car();” but not the line that called “speed” so that I could analyze that.

Here we see that assembly is creating space:

```
# BB#1:
        mov     eax, 8
        mov     edi, eax
        call    _Znwm
        xor     esi, esi
        mov     ecx, 8
        mov     edx, ecx
        mov     rdi, rax
        mov     qword ptr [rbp - 24], rax # 8-byte Spill
        call    memset
        mov     rdi, qword ptr [rbp - 24] # 8-byte Reload
        call    _ZN1AC2Ev
        mov     rax, qword ptr [rbp - 24] # 8-byte Reload
        mov     qword ptr [rbp - 16], rax
        jmp     .LBB1_3
```

This is how both calls set up their function calls. Here we see it creating space for the virtual method table. In this it will insert the standard parent class A object, which we see it doing in the allocated memory. Later one, once the method is being compiled and then run, it will update the object with the necessary information on where it will be getting its data on. This it does later on at a separate memory space that checks the conditions met on the parameter or meant to be applied.

Here we see where the dynamic dispatch actually occurs in the speed function call:

```
.LBB1_3:
        mov     rax, qword ptr [rbp - 16]
        mov     rcx, qword ptr [rax]
        mov     rdi, rax
        call    qword ptr [rcx + 8]
        xor     edx, edx
        mov     dword ptr [rbp - 36], eax # 4-byte Spill
        mov     eax, edx
        add     rsp, 48
        pop     rbp
        ret
```

The first two move lines function as a pointer to where the object is stored in memory. The first one goes to the memory location that has a pointer in it, “[rbp - 16]”, and moves that into rax. Then we go to the value specified by that memory address, which contains the memory address [rbp - 16], and moves that into rcx. Rcx is now a functional pointer that points to the top of the virtual method table. After that, since speed is the second method defined, it has to go to the second memory address in the virtual method table. In order to do this we simply see it adding 8, 8 being the number of bytes used to store a memory address, to rcx. Since rcx is holding the location for the top of the VMT, by adding 8 we go to the second address in the table, which happens to be the location of

the function call for the speed object. Since we are just going to the object and we don't know what it will be, the computer doesn't figure out which actual function code it will respond with until runtime. At runtime, it will go to the speed function defined in class A or the speed function in class B depending on which condition is met at runtime. Here we see the code that will execute for either method call:

The code for A:

```
.Ltmp17:
        .cfi_def_cfa_register rbp
        mov     eax, 1
        mov     qword ptr [rbp - 8], rdi
        pop     rbp
        ret
```

And then for B:

```
.Ltmp23:
        .cfi_def_cfa_register rbp
        mov     eax, 2
        mov     qword ptr [rbp - 8], rdi
        pop     rbp
        ret
```

Here the actual function runs depending on if it goes to A or B, and a return value is stored into eax and then returns. The value in eax is what then gets printed out in a cout statement by the end of the code and you have the final answer from that.

Citations:

Assembly instructions:

- https://docs.oracle.com/cd/E19455-01/806-3773/instructionset-27/index.html
- http://www.c-jump.com/CIS77/ASM/Flags/F77_0160_sar_instruction.htm
- https://en.wikibooks.org/wiki/X86_Assembly/Shift_and_Rotate
- http://stackoverflow.com/questions/13223756/assembly-arithmetic-right-shift-sar
- https://wiki.skullsecurity.org/Simple_Instructions
- http://stackoverflow.com/questions/2987876/what-does-dword-ptr-mean
- http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
- http://www.c-jump.com/CIS77/ASM/Instructions/I77_0250_ptr_pointer.htm

C++ Dynamic Dispatch:

- http://stackoverflow.com/questions/25204803/methods-of-dynamic-dispatch
- http://condor.depaul.edu/ichu/csc447/notes/wk10/Dynamic2.htm
- http://members.gamedev.net/sicrane/articles/dispatch.html