Question 2

a) Write a version of the function using a for loop

```
long forsum (long start, long finish)
{
    long acc = start;
    long i;

    for (i = start+1; i <= finish; i++)
    {
        acc += i;
    }
    return acc;
}
```

b) Write a version of the function using a while loop

```
long whilesum (long start, long finish)
{
    long acc = start;
    long i = start+1;

    while (i <= finish)
    {
        acc += i;
        i++;
    }
    return acc;
}
```

c) Write a version of the function using a do loop

```
long dosum (long start, long finish)
{
    long acc = start;
    long i = start+1;

    do
    {
        acc += i;
        i++;
    } while (i <= finish);
    return acc;
}
```

d) Write a version of the function using a goto loop

```c
long gotosum (long start, long finish)
{
    long acc = start;
    long i = start +1;

    loop:
        acc += i;
        i++;
        if (i <= finish)
            goto loop;
    return acc;
}
```

e) Is the assembly language version of each loop function the same or different? If different, identify the differences. Your comparison should be based on:

- Number of registers used
- Number of jumps (iterations)
- Total number of operations

The following is the assembly code for the for loop and the while loop.

```
3   _forsum:
4   LFB4:
5       movq    %rdi, %rax      ; long acc = start                    note: acc in %rax and start in %rdi
6       leaq    1(%rdi), %rdx   ; long i = start + 1                  note: i in %rdx
7       cmpq    %rsi, %rdx      ; compare i to finish --> guard       note: finish in %rsi
8       jg  L1                  ; guarded do                          if i > finish, return.
9       addq    $1, %rsi        ; finish++ (??)
10  L3:
11      addq    %rdx, %rax      ; acc += i
12      addq    $1, %rdx        ; i++
13      cmpq    %rsi, %rdx      ; compare i to finish
14      jne L3                  ; if finish != i, jump to L3          !! whilesum and forsum are exactly the same.
15  L1:
16      ret
```

```
31  _whilesum:
32  LFB6:
33      movq    %rdi, %rax      ; long acc = start                    note: acc in %rax and start in %rdi
34      leaq    1(%rdi), %rdx   ; long i = start + 1                  note: i in %rdx
35      cmpq    %rsi, %rdx      ; compare i to finish --> guard       note: finish in %rsi
36      jg  L9                  ; guard                               if i > finish, return.
37      addq    $1, %rsi        ; finish++ (??)
38  L11:
39      addq    %rdx, %rax      ; acc += i
40      addq    $1, %rdx        ; i++
41      cmpq    %rsi, %rdx      ; compare i to finish
42      jne L11                 ; if finish != i, jump to L3          !! whilesum and forsum are exactly the same.
43  L9:
44      ret
```

We see here that the assembly code of the for loop and the while loop are identical. Both use exactly four registers and store: acc in%rax; start in %rdi; finish in %rsi; i in %rdx. Both loops have a guard (on lines 8 and 36, respectively) with conditional jump instructions that cause the loop to iterate zero times if start > finish.

In the event that the guard test fails and the loop does iterate, the compiler makes a slight tweak to the C code by incrementing finish (lines 9 and 37), then uses a do… while loop with a test that queries whether i != finish. This is similar to iterating while i < finish in C code, which is functionally the same as how the code was written since finish has been incremented.

The total number of instructions executed are 5 if start > finish, and a minimum of 10 if start <= finish. Each iteration requires 4 instructions, so the number of instructions executed can by expressed as

x = 5, if start > finish
x = 6 + 4(n), where n = number of loop iterations, if start <= finish.

The following is the assembly code for the do loop and the goto loop.

```
17  LFE4:
18      .globl _dosum
19  _dosum:
20  LFB5:
21      movq    %rdi, %rax      ; long acc = start                  note: acc in %rax and start in %rdi
22      leaq    1(%rdi), %rdx   ; long i = start + 1                note: i in %rdx
23  L7:
24      addq    %rdx, %rax      ; acc += i
25      addq    $1, %rdx        ; i++
26      cmpq    %rsi, %rdx      ; compare i to finish
27      jle L7                  ; if i <= finish, jump to L7        !! goto1sum and dosum are identical
28      ret
```

```
45  LFE6:
46      .globl _goto1sum
47  _goto1sum:
48  LFB7:
49      movq    %rdi, %rax      ; long acc = start                  note: acc in %rax and start in %rdi
50      leaq    1(%rdi), %rdx   ; long i = start + 1                note: i in %rdx
51  L15:
52      addq    %rdx, %rax      ; acc += i
53      addq    $1, %rdx        ; i++
54      cmpq    %rsi, %rdx      ; compare i to finish
55      jle L15                 ; if i <= finish, jump to L15       !! goto1sum and dosum are identical
56      ret
```

We see in these two cases that the assembly code of the do loop and the goto loop are also identical. Both use exactly four registers and store: acc in %rax; start in %rdi; i in %rdx, and finish in %rsi.

Neither loops have a guard and so both loops are guaranteed to iterate at least one time. As such, functionally these two procedures will give different answers from the for and while loops for instances where start >= finish. For example, if start = 5 and finish = 0, both the do and goto loops written here will return a value of 11, while the for and while loops return an answer of 5. It is fair to say that this does not represent the intention of the programmer in this sense.

With respect to the actual assembly code, the compiler very faithfully represents the C code in both cases, with the instructions being near identical representations of what was written in C.

Both of these procedures execute a minimum of 7 instructions due to the fact that they guarantee at least a single execution. Each additional loop iteration requires another 4 instructions.

The number of instructions executed can be expressed via

$$x = 3 + 4(n), \text{ where } n = \text{number of loop iterations and } n >= 1$$

It is worth noting that the goto loop has various implementations, and it just so happened that the version I wrote mirrored the do loop. The use of goto code gives the programmer utmost control, and so it would be possible to write a goto loop that mirrors the assembly code of the for and while loops if so desired (by including a guard with a goto label that returns the function prior to iterating, for example).