Question 1.

Method 1:

Miss Rate = Misses/Requests = (Struct Size/Cache Size)/Requests

All of the excess data that we are not using is pulled into the cache when we access each airplane's position and nearest airport position. Therefore, we take the entire size of the struct (40 bytes) and the size of our cache line (64-byte) and plug the values into our formula. Additionally, we know we make 2 requests per iteration.

Miss Rate = Misses/Requests = (40*N/64)/(2N) = 0.3125

This method has a higher cache miss rate, but is much more intuitive to use. Therefore, it is likely that we will choose this implementation if our N is small and impact on performance is negligible.

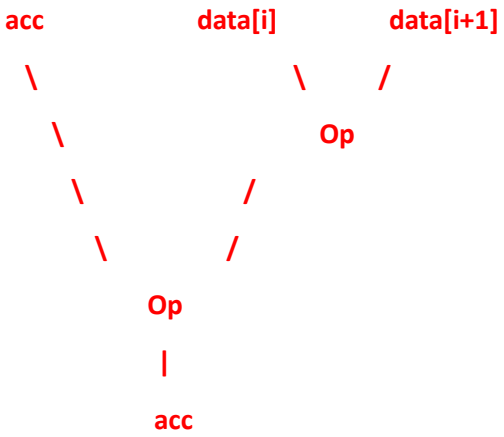
Method 2: By simplifying our struct into JUST the x and y float values to act as a sort of "parent" struct, we can create an individual struct array for each value type we want to store. This simplifies our operations: each float is 4 bytes, so we can fit 16 into our cache. Each iteration accesses 2 bytes, which fits evenly into our cache. This means we only miss the first access of every 16 floats.

Miss Rate = Misses/Requests = 1/16 = 0.0625

While this is more optimized, it may be less intuitive to work with and is therefore better to use if N is extremely large.

Question 2.

a. **The critical path for a single loop is going to be the longest path. Looking at our data flow diagram below, the critical path would be the load of our data, the op between our loads, then the op with acc**



b. **The critical path is always going to be from one register back to itself, or one variable back to itself for loops. In this case, we are dependent upon acc. Acc relies on the operation between acc and (data[i] op data[i+1]). Data[i] and data[i+1] are not dependent, so the critical path is acc performing op with the other value, and storing it back into acc.**

c. **We have incorporated a 2x1 loop unrolling, and we are solely dependent upon acc waiting on acc from previous loops. Unrolling it again to be a 4x1 loop unrolling, we would half our latency bound again.**

   You could also create a second accumulator, creating a 2x2 unrolling, or even a 4x2 unrolling, which would shrink the latency bound as well.

Question 3.

1D Stencil

(a) 1/12

- Each cache line holds 4 doubles (32 bytes)
- 2N iterations in total, from the nested loops
- `Kern` is brought into cache once, and reused in each iteration
    - approximately 0 miss per iteration
- For the array `Out`, 1 miss every time the outer loops counter `i` increments by 4
    - approximately 1/8 miss per iteration
- Similarly for the array `In`
    - approximately 1/8 miss per iteration
- Per iteration, (0 + 1/8 + 1/8) = 1/4 miss
- 3 memory accesses per iteration
- miss rate = # cache miss / # memory access = (1/4) / 3 = 1/12

(b) The resulting `Out` array is [1.5, 2.5, 3.5, …]

- Array `In` is initialized as [1, 2, 3, 4, …]
- `Kern` is [0.5, 0.5]
    - This stencil/kernel just averages two neighboring elements
- Note: this question has little to do with memory organization or performance optimization. The aim is to use a toy example to hopefully provide some intuition about the stencil computation in the performance lab.

2D Stencil

(a) 5/192

- Each cache line holds 4 doubles (32 bytes)
- 16 (N^2) iterations in total
  - from the four nested loops N x N x 4 x 4
- `Kern` is brought into cache once, and reused in each iteration
  - approximately 0 miss per iteration
- For the array `Out`, 1 miss every time the loops counter `j` (the second outmost loop) increments by 4
  - approximately 1/64 miss per iteration
- For the array `In`, 4 miss on average when the loops counter `j` (the second outmost loop) increments by 4
  - brings in 4 cache lines that respectively start with the addresses: `In[i][j], In[i+1][j], In[i+2][j], and In[i+3][j],`
  - approximately 1/16 miss per iteration
- Per iteration, (0+ 1/16 + 1/64) = 5/64 miss
- 3 memory accesses per iteration
- miss rate = # cache miss / # memory access = (5/64) / 3 = 5/192

(b) This is more of an open-ended discussion question.

- First of all, before considering how, why reducing miss rates?
  - Compared to cache hit, cache miss is expensive (order of magnitude more clock cycles)
- Use tiling/blocking to improve spatial locality
  - The total cache size is less than 8N, so it is not large enough to even hold a full row of data (of either the input or the output array)

```
/* an example of blocking (tiling) */
const int TILE_LEN = 64; // TODO: this value needs to be tuned
for (int ii = 0; ii < OUT_LEN; ii += TILE_LEN)
      for (int jj = 0; jj < OUT_LEN; jj += TILE_LEN)
            for (int i = ii; i < ii + TILE_LEN; i++)
                  for (int j = jj; j < jj + TILE_LEN; j++)
                        for (int x = 0; x < KERN_LEN; x++)
                              for (int y = 0; y < KERN_LEN; y++)
                                    Out[i][j] += In[i+x][j+y] * Kern[x][y];
```