**Testing**

While testing for the time complexity of our program (The Carry Look Ahead Adder) we had swapped from dynamic initialization to static initialization so as to see the effect of large malloc on the time complexity. After knowing that the difference of the time between static and dynamic initialization (which was almost nil) we again swapped our program back to dynamic initializations.

During the above process we had produced a bug which didn't seem familiar to us. It was a run time error stating

**\*\*\*\* stack smashing detected \*\*\*: ./a.out terminated\***

Briefly, stack smashing occurs when a function allocates a static array on the stack and writes past the end of it, onto other local variables and eventually onto other function stack frames. When it comes time to return from the function, the return address has been corrupted and the program ends up some place it really shouldn’t. In the best case, the program just crashes; in the worst case, a malicious party crafts code to exploit this malfunction.

We found that the above error is due to one of the methods of Buffer overflow protection.

The compiler we used for compiling our C programs is GCC. GCC comes with many security measures and SSP(stack smashing protection) being one of it.

The program after debugging has been tested with lots of varieties of data sets and it has executed all of them without any garbage result.

**Discussion**

Our project was initially about making adder that adds any large integers using the circular linked list. Basically in the basic level implementation these adders are implemented using the shift registers which can exactly be implemented using the circular linked list.

In our discussion about ripple-carry adders, we said that adding two n-bit numbers requires O(n) time. The reason is the carry. As you perform the addition on the least significant bits, you may have a carry that "ripples" its way to the most significant bit. If we didn't have a carry, then we should be able to add in O(1), because adders can work in parallel.

Since we already had learnt about the carry look ahead adders we wanted to implement it in decimal so as to compare the working of both and its time complexities.

The carry look ahead logic requires a special hardware support so as to allow it to compute the carries simultaneously.

The implementation of the carry look ahead adder uses parallel programming which is achieved using the **openmp** library for C.

The carry generation is done using a recursicve relation..

There are two functions generated. G and P. They are computed as follows,

**Gi = (Xi + Yi) / 10**

**Pi = (Xi + Yi) % 10**

Using the above two values the carry and sum is produced.

**Ci = Gi | (Pi > 9) ? Ci-1 : 0 and C0 = 0**

**Si = (Pi + Ci) % 10**

By using these formulas, we can cut down the adders from O(n) to what appears to be O(1) time. After all, pi and gi only depend on xi and yi, which are the bits of x and y which are immediately available to us (Parallel computation) . They also only depend on c0, which is also immediately available. We don't have to wait for carries to perform this computation.

Still, it's not quite O(1). Why not?

After achieving the parallelism using OpenMP we understood many cons of software level parallel programming. The time complexity was hugely affected after the use of parallel programming but it was the opposite extreme to what we expected. When we compared the time complexity with various set of values for n(number of digits) for the serial and fast adder, we found that the serial adder was way faster initially( in the order of 50). We were confused initially but made the following conclusion after analising the working of parallel programming:

1. The amount of time taken to create and synchronize the thread had a lot of effect on the time complexity.
2. The actual number of threads that works simultaneously depends on the system architecture, (in our case no of threads = 4, quad core processor). Even though we are able to calculate all the carries in parallel the hardware limits the number of parallel computations.
3. Since the data is shared between the threads, shared memory model is used, which is again an overhead for the computation.

The theoretical value of the time complexity of the serial adder is O(n) and pratically too, it behaved the same way.

The theoretical value of the time complexity of the carry look ahead adder is O(log(n)), but practically it was k1 \* O(log(n)) + k2 , where k2 is very large(overhead of creating and managing thread). But it can be said that on really huge data the time complexity can approach its theoretical value.