Final Project: Fraig b03901036 陳柏文

I. Function Explanations:

1. **CirMgr**:

   I use an GateList defined in the CirDef.h to store all the gates as pointers. And we can access the gate by the id of the gate which corresponding to the index of the GateList. (Ex: _gateList['gateID']). The time for accessing gates is constant time. To connect gates, I store another IdList to store the ID of fanIN. I also store _dfsList and _undefList to handle the problem and function below.

2. **CirSweep**:

   I implement CirSweep by storing a bool member data in CirGate class to indicate whether each gate is in the depth search list. If it's not in the DFS list, we will delete it. The destructor of the gate will automatically disconnect it from its fanIN and fanOUT.

3. **CirOptimize**:

   I implement the four cases in the spec. And for each cases, we all end up in a **replacement function** to replace gate a by gate b. The function takes three arguments, a pointer of the gate(a) to be replaced, another pointer of the gate(b) to replace, and a Boolean to indicate whether these two gates are the inversion of each other. And what this function do is that it reconnects the fanOUT of gate a to gate b and disconnect it from its fanIN.

4. **CirStrash**:

   The purpose of this function is to merge two gates (Gate A and Gate B) with the same fanINs. First, B's fanOUT gates shall change their input fanin IDs to the literal ID of A based on the sign of their connection. Second, B's fanIN gates shall delete their fanout to B and reconnect to A. Lastly, B's fanOUT gates shall be pushed into A's fanout list. In this function, a HashMap is used to store the keys and the CirGate pointers. Since there are two fanIN, they are combined into a size_t to form a key. This calculation can be done in constant time, and the key value is guaranteed to be much larger than the hashmap bucket size. The performance of this function is quite satisfying, so that this scheme may effectively reduce the number of collisions.

5. **CirSimulate**:

Parallel pattern encoding is used in this function. Instead of using unsigned to store the pattern, I use size_t to store the patterns, and then we can use bitwise operation to calculate the simulation values for all gates. A HashMap is used to store the pairs. However, the insert function is different from the optimize function version; it won't allow collision to happen, and when a new node with different key value needs to be inserted, it will be stored into a separate vector instead.

II. **Analysis of experiments**:

For the basic functions such as sweep(), strash(), and optimize(), the output and error message of my function is fine tuned to exactly match the reference program. Runtime is compatible, yet the memory usage is slightly higher than the reference program. My Simulation program works fine on typical circuits, but it can be significantly slower than the reference program when large circuits such as sim13.aag is optimized. I believe a better data structure to handle collision gates can be really helpful in improving its efficiency.

Below is implemented on sim13.aag

Memory/time	My fraig	Ref fraig
cirread	35.48 MB/0.1 s	13.72 MB/0.07 s
cirsweep	38.28 MB/0.01 s	17.72 MB/0.02 s
cirstrash	38.28 MB/0.01 s	17.72 MB/0.00 s
ciroptimize	39.78 MB/0.02 s	17.72 MB/0.00 s