DSnP Final Project

FRAIG

學號：B01901056

姓名：張凱崴

系級：電機三

手機：0983063316

Concepts:

This project is composed of five parts: READ, OPTIMIZE, STRASH, SIMULATION and FRAIG.

The first part, READ is completed in HW6, so in the following of the report, I will explain my work on the other four parts in order.

Practices:

1. Optimize

In this part, we want to simplify the circuit by four kinds of trivial optimization: (1) AND gates with a const 0 fanin, (2) with a const 1 fanin, (3) with the same fanins and (4) with inverse fanins.

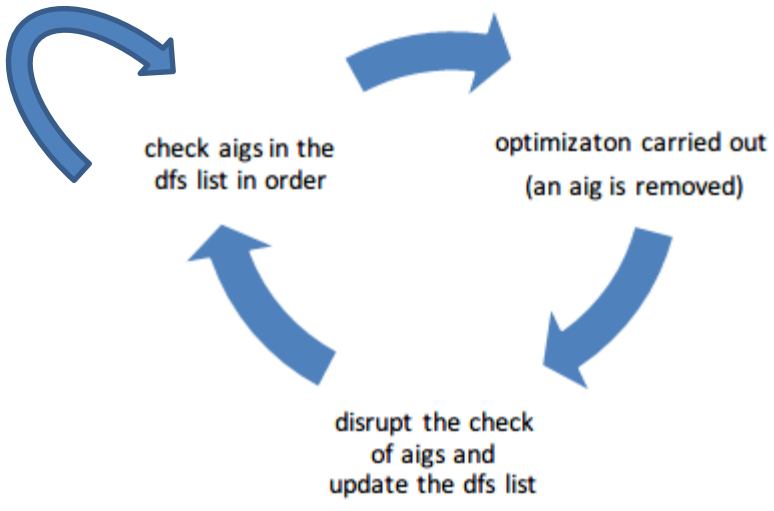
After observation, all four ways are relevant only to **the remove of aig gates**, and when removing a gate, there are at most three steps included:

(1) remove the corresponding fanout of aig's fanins

(2) change the fanin of aig's fanouts to another aig of pi

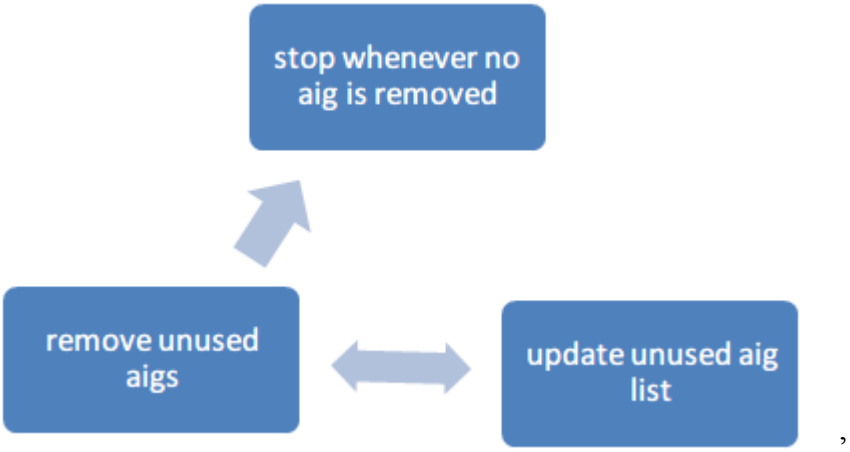
(3) transfer aig's fanout to another aig of pi

Therefore, I impliment three functions (they are member functions of aig, pi, po, const, undef gates) to deal with these three steps. And in the four cases, I just need to decide which gate to call which functions. In order to do optimization efficiently and correctly, I always conduct optimization to the aig gates in the order of dfs list (from PI to PO). However, when an aig is removed, there may be some gates becoming unused. And in some special cases, if the unused gates are original in the dfs list, there will be some problems to handle during the optimization. So I choose to update the dfs list everytime I remove an aig gate. The work flow is showed below:



The optimizaton is always carried out to the aigs in the dfs list. But in order to keep the circuit clean, we often do SWEEP before optimization, which means to sweep out all aigs that cannot be reach from po (not in the dfs list).

To keep the problem simple, I don't want to detemine what gates should be removed in one step. Because I think it's complicated when removing no use aigs and considering their fanins and fanouts, though it may be easy to determine with the help of dfs list. Anyway, I decided to do it step by step:



,

As a result, the order of removing the not-in-the-dfs-list gates is different from the ref program. But the final result is the same.

2. Strash

In this part, we need to find out the aigs that are structually equivalent, which means aigs with fanin A&B and B&A.

If we compare each of the gates one by one, we will suffer the **time complexity $O(n^2)$**. So I use the HashMap to classify them into buckets. Gates in the same buckets means their HashKey() have the same value when moduloing _numBuckets. And in order to make gates with fanin A&B and B&A have the same HashKey(), I defined the HashKey to be **a class with the two fanins (in poiner form) of the gate as its data member** and HashKey() to be "**size_t(fanin1)*size_t(fanin)"**. In this way, the HashKey() is supposed to be very different from that with different fanins because of the multiplication operation.

After the classification by HashMap, the number of comparison of the gates will be much fewer. So we can compare them one by one. Same as OPT, whenever a pair of structual equivalent gates is found and one of the aig gates is merged (removed), I disrupt the comparison and restart the Hash process again for fear that some problems may occur. Repeat the steps until no equivalent gates are found.

3. Simulation

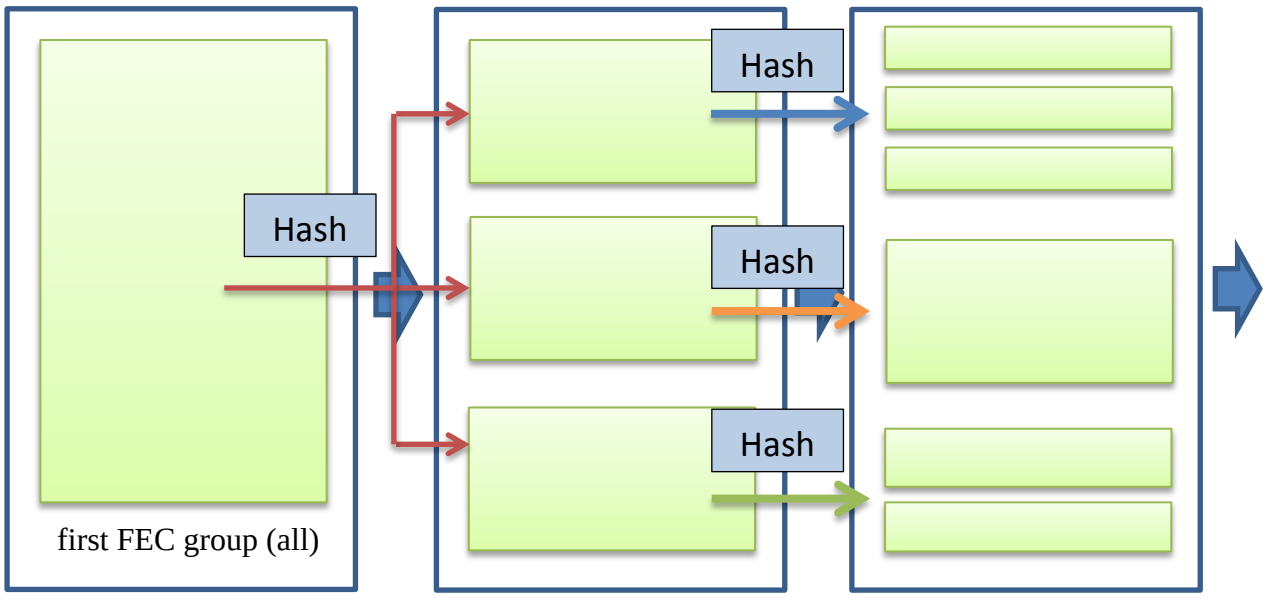
In this part, we need to give simulation values to all pi and aig gates, and divide them (const 0 and aig only) into FEC groups.

First of all, I set a data member in every gate to record the 32-bits simulation value with a size_t data type. The conversion of binary and decimal value is handled by a global function and only used when printing on the screen or importing from pattern files. Then the aig gates simulation values can be detemined with the two fanins' simulation value by bitwise-and operator (&). We should note that the simulaton order is accordance with the dfs list. Only pi, aig and po gates will have changing simulation values. Other gates are set to be 32-bits 0 all the time.

After the simulation, we need to compare the simulation value of const 0 and aig gates and then divide them into several fec groups. Same problem as the STRASH, if we compare them one by one, we will suffer **O(n^2) complexity**, so I used HashMap once again. I defined the HashKey to be **a class with the simulation value of the gate as its data member**. In order to find not only the FEC groups but also the IFEC groups, I defined the HashKey() to be **"(simValue) * (~simValue)"** so that the gates with totally inverse simulation values will be classified into the same bucket. Then we can do the on-by-on comparison to the gates in the same buckets now.

If the simulation values are the same or totally inverse, put the gates together into a fec group. And we will have several fec groups after one simulation (32-bits). Then repeat the same process until it "fails too many times" (a "fail" means a simulation that produce no new fec groups.) in a **random simulation**, or the simulation pattern is over in a **file simulation**.

The work flow is showed below:



In the random simulation, we need to think of algorithms to determine:
(1) **the random pi simulation values**
(2) **the maximum fail time of simulation.**

Theoritically, **each bit of the random pi simulation values should have the same possibility of 0 and 1** so that the simulation is random enough to test the function of the whole circuit. Secondly, I think the maximum fail time should be relevant to **the number of fec groups** and **the number of the gates in each fec group**. I will discuss this two problems further and show how I decided my algorithms in the next topic.

4. Fraig

I didn't finish this part.

Algorithm:

1. Random simulation value

Mentioned above, I wish to make every bit of simulation as random as possible, so I use the random number generator in the director "util". However the data type the random number generator produce is 'int', which is 16 bits. Therefore, I use << operator to help generate a random unsigned int:

```
size_t(rnGen(INT_MAX))+(size_t(rnGen(INT_MAX))<<16)
```

2. Mgaximum fail time

I think fail time is relevant to two things: **the number of fec groups** and **the number of the gates in each fec group.** Because if the larger number of fec groups, the more complicated the circuit is, generally. So the fail time should be more. And we prefer dividing a fec group with more gates to a group with fewer gates. So the larger number of the gates in one group, the fail time should be more, Therefor, I set the fail time by the function below:

```
size_t fail_time = log10(total_gate_number);
for (size_t i=0, n=fec_groups.size(); i<n; ++i)
   fail_time += log(fec_groups[i].size());
```