

DSNP final report FRAIG

學號：b04507025

系級：電機三

姓名：韓秉勳

email: b04507025@ntu.edu.tw

一、主要架構：

1. CirGate member 介紹：

每個 gate 我採用繼承的方式生成了 PI GATE , PO GATE , AIG GATE , CONST GATE , UNDEF GATE 。

Gate member 成員如下：

```
size_t _ref;  
string _name;  
unsigned _line;  
unsigned _gateid;  
GateType _type;  
bool dflag;  
vector<CirGateV> _faninList;  
vector<CirGateV> _fanoutList;  
size_t _smiupos;  
bool _fehead;  
Var _raigvar;  
bool _raigdel;  
size_t _simpattern;
```

_type , _gateid , _line 分別記載邏輯閘的型態，id 與 aag 檔中的行數，_ref 跑 depth first search 使用), dflag 判斷 gate 是否在 dflist 中，以及為了方便以 vector 儲存的 faninList , fanoutList，裡面存的是 CirGateV 的物件，記載了反向的資訊。

在 simulation 與 raig 方面，我也另外存了一個 gate 在 fecgroup 中的 position(_smiupos) , simulation 會用到的 pattern, 以及 SAT 的變數(raigvar) 等等。儲存這麼多變數是為了能讓操作方便，以及使用空間換取時間，所以當使用有 80000 邏輯閘的 sim13.aag 讀進來的時候時間如下：

```
fraig> cirr sim13.aag  
  
fraig> usage  
Period time used : 0.07 seconds  
Total time used : 0.07 seconds  
Total memory used: 31.26 M Bytes
```

而 ref 是這樣：

```
fraig> usage  
Period time used : 0.03 seconds  
Total time used : 0.03 seconds  
Total memory used: 13.57 M Bytes
```

時間上其實蠻快的，只是記憶體用量多了很多。

2. CirMgr member 介紹:

```
private:
    size_t _total;
    size_t _innum;
    size_t _latnum;
    size_t _outnum;
    size_t _andnum;
    ofstream * _simLog;
    vector<CirGate*> gatelist;
    //vector<unsigned> idlist;
    vector<unsigned> inlist;
    vector<CirGate*> outlist;
    vector<CirGate*> dflist;
    vector<vector<CirGate*> > fecgroup[2];
    bool dfupdate;
};

#endif // CIR_MGR_H
```

我存了一個 **gatelist** 代表總表，還有 **outlist**，**inlist** 分別代表 PO PI，另外還有 **dflist** 來存 dfs traverse 到什麼。較為特別的是我的 **fecgroup** 兩個二維 **vector** 組合而成的 **array**，這是為了讓 **updatefec()** 更為方便，在 **simulation** 會詳細說明。

3. Hashmap 設計:

基本設計與 **hashset** 相同，但多了設計 **key** 的部分。在 **key** 的設計上，我做了 **Hashkey** 與 **Simkey** 兩種 **class**，分別針對 **CirStrash** 與 **CirSimulate** 使用。

```
44
45 class HashKey
46 {
47 public:
48     HashKey(size_t a , size_t b): in0(a) , in1(b) {}
49     size_t operator() () const {
50         return (in0 | in1) + (in0&in1) ; }
51     bool operator == (const HashKey& k) const {
52         if(in0 == k.in0 && in1 == k.in1) return true;
53         if(in0 == k.in1 && in1 == k.in0) return true;
54         return false;
55     }
56
57 private:
58     size_t in0 ;
59     size_t in1;
60 };
61
62 //
```

Hashkey 如圖，存的是每個 **AIG gate** 的兩個 **fanin**(**in0** , **in1**)，分 **hash** 時會讓兩個 **fanin** 進行 **bitwise or** 和 **bitwise and operation** 複合操作，讓每次 **gate query** 時 **hash** 能找到自己跟哪一個 **gate** 最像。

```

24 //template <class Key>
25 class SimKey
26 {
27 public:
28     SimKey(size_t a ): pattern(a) {}
29     size_t operator() () const {
30         return (pattern) * ~(pattern) ; }
31     bool operator == (const SimKey& k) const {
32         if(pattern == k.pattern ) return true;
33         if(pattern == ~(k.pattern))return true;
34         return false;
35     }
36     size_t getpattern(){return pattern;}
37
38
39 private:
40     size_t pattern ;
41
42
43 };
44

```

Simkey 如圖，則是讓 simulate pattern 進來能分越散越好，故我把 simulate pattern 乘以他的反向作為 Hash 分組的指標(這樣做可以讓結果較亂，除非很像的才能擺在一起)，使一個 pattern 能盡快辨識出與他相同的 pattern。

二、project 函數實做簡述:

1. CirSweep:

在每個 gate 都設一個 dfs 的 flag，只要沒設 flag 且非 PI、PO 就把 gate 去除。

2. CirOptimize:

將 4 種條件(接到 CONST0,CONST1,兩個 fanin 一樣 or 反向)分別列出，將接到 CONST0 的 AND gate 與 fanin 都不一樣的 AND gate 以 0 表示，接到 1 的相同者以另一個 fanin 表示，兩個 fanin 一樣的以一個 fanin 取代，最後再創 mergegate 的函數把 fanin，fanout 接好。

3. CirStrash:

每個 gate 利用 hash 去 query HashKey 相同的組別，若真的是 fanin、fanout 相同就做 merge，不同就放進 hash。因為有這動作，strash 速度頗為快速。

我 hashsize 的大小開的是 gatelist 的大小,因為 strash 操作只要比對 dflist 中的就好，複雜度屬於 $O(n)$ ，將 size 開大有利於分組的正確性，如圖：

```

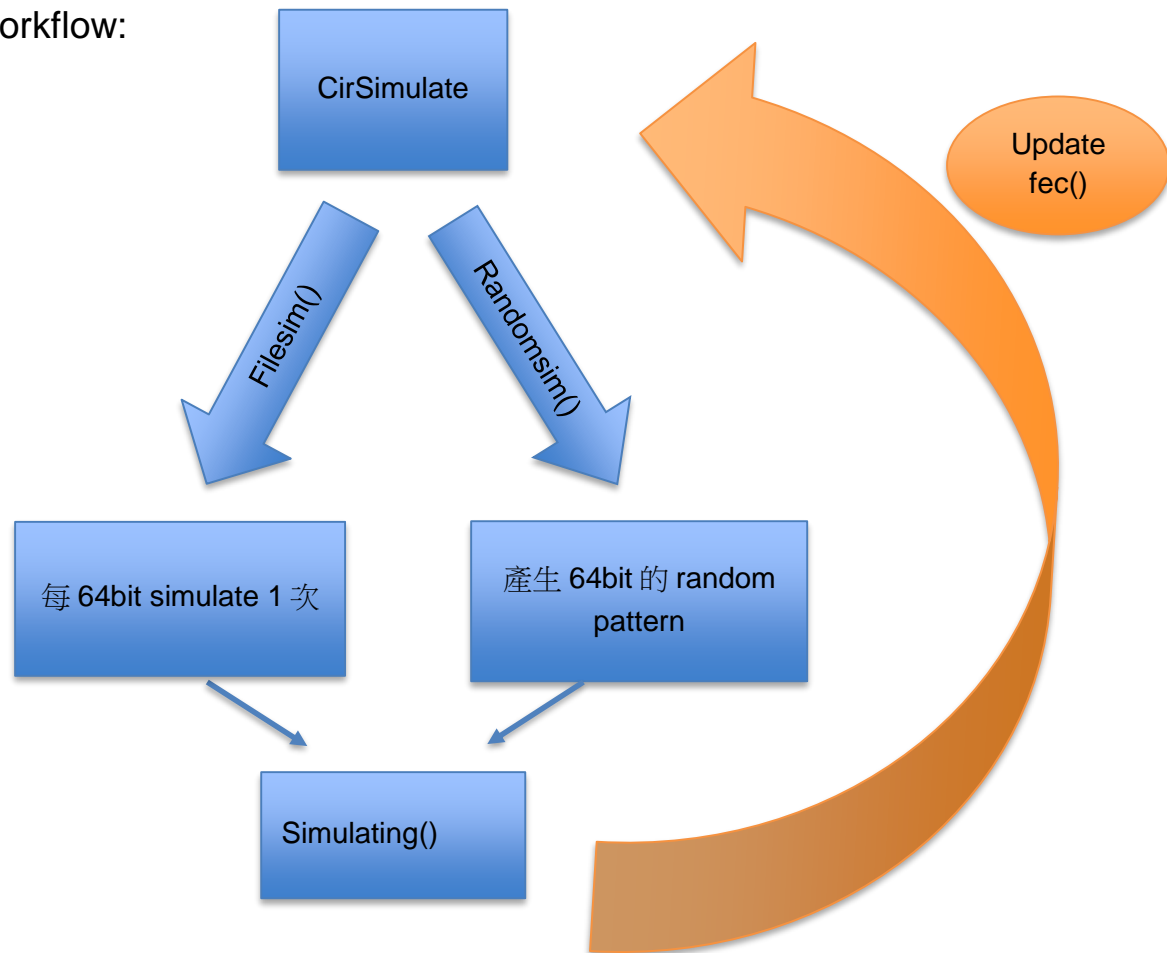
HashMap<HashKey , CirGate*> Hashmap(gatelist.size());

```

我有遇到 1 個 gate strash 完接到相同 fanout 的情況，但比對 reference code 看來是 Optimize 的工作，strash 不去處理。

4. CirSimulation:

a. workflow:



b. 詳述架構:

(1). Filesim():

用一個與 PI 一樣多的 vector 儲存每個 PI 的 simulate pattern。每讀一排就把原本的 data shift 一次與讀到的東西進行 Bitwise or，並在讀到 64 倍數行時進入 simulating()。若最後面不足 64 行的一樣也會丟進 simulating()，即可達到讀檔的功效。

(2). Randomsim():

將 rnGen()轉成 size_t 並 shift 32 位，再與另一個 rnGen() bitwise or 即可生成 64bit 的 input pattern，cout 出來的 key 大概是這樣：

```

randomkey: 01101100_11101000_11010001_00111111_00010110_11110010_00011111_01101110
randomkey: 00100101_11111100_11111100_11110010_01100011_01110011_00111011_01010011
randomkey: 00001110_00111010_11001011_01100000_00110000_00000111_01011110_00111010
randomkey: 00010101_01010011_00010010_10010111_01111011_11111110_10110010_01010011
randomkey: 00110110_11110010_01111011_01101110_01100010_10011001_01111101_01111000
randomkey: 01011111_10101000_11101111_00010111_01110100_10100010_10010011_01011111
randomkey: 01110100_01000010_10111000_01111100_00011001_11010111_10110010_10100100
randomkey: 01101010_11111110_10000100_10111111_01111000_00011001_01010100_10111101
randomkey: 01100001_10000000_11000011_01100000_00101101_01010111_01100011_00110001
randomkey: 00010001_00010010_00101100_10111010_01101011_00000111_01101011_11101110
randomkey: 00111111_00011011_11101011_11111001_00111100_01100100_11001010_10010110
randomkey: 01111110_11111110_00101101_10111000_01010101_10111101_10000111_10111111
randomkey: 01010001_01000001_10010010_01110010_01000111_11000110_11101011_10101011
randomkey: 01000011_10100100_11100100_11011111_00101101_00000111_10100110_11100110
randomkey: 01101101_01111000_10010101_11010110_00110111_10101111_01110011_00001100
randomkey: 00010100_01100000_11000011_00100000_01011010_01100001_01100111_00010110
randomkey: 01001110_10100001_10010010_01111011_00111010_01011101_11000000_00010010
randomkey: 00111101_11010100_10100010_01101001_01011100_11011100_01011101_11011011
randomkey: 01101010_01100101_00011110_01001101_01010011_00100111_10110101_00000000

```

看起來非常無規律，頗符合需要的。

(3). Simulating():

把 64 bit pattern 餵給 PI ,然後利用 recursive call 從 PO 利用 dfs 方式將 pattern 傳至每個 gate ，再用新的 pattern 將 fecgroup 更新。

另外若進到 simulating 時 fecgroup 完全沒有 gate ，則需要將全部 dfslist 的 gate 都丟進 fecgroup 第一排當中，再根據 pattern 進行分組。

(4) updatefec()

大致流程如下：

暫存 fecgroup 清空

for(所有原本的 fecgroup){

 每行原本的 fecgroup 產生一個 hashmap

 產生每個 gate 的 Simkey

 if(hashmap 中有一樣 pattern){

 放入該暫存 fecgroup 對應的列}

 Else{暫存 fecgroup 新增一列，放入 gate}

}

 交換(原本 fecgroup, 暫存 fecgroup)

簡言之一開始 CirMgr 的 fecgroup 需要 2 個(主要的 fecgroup 、暫存 fecgroup)就是為了 update 時能簡單操作，因為常常會把 fecgroup 中的 Member 增減，不好 maintain，乾脆直接令另一個暫時的 fecgroup 把更新的都放進暫時的 group，最後再交換位置就可以了！

Hash 在這裡的功用就是稍微先依照 **key** 分一點組，一模一樣的 **pattern** 只會有一個 **gate** 在 Hash 中。每次有 **gate** 是相同的 **key**，就去看 hash 裡有沒有相同的 **gate**，若有就放入原本 hash 中的 **gate** 對應的行數，沒有就塞到新一排的 **fecgroup**。

其中 **fecgroup** 開的 hash 的 **size()** 與 **simulate** 的時間頗為相關，為此我做了一次比較：

hash size 與時間關係（測 sim13.aag / pattern.13）

Hash 的 size()	時間	記憶體使用量
3*每行大小	3.6s	35.5M
5*每行大小	3.69s	39.5M
7*每行大小	3.89s	42.83M
10*每行大小	4.06s	48.5M
15*每行大小	3.94s	57.83M
20*每行大小	3.96	67.47M

由此圖可知在 **hashsize = 3** 時有最好的時間與記憶體表現，但因 **hashsize** 開太小會使分組難以分出，取捨後選擇以 **hash size=5** 作為最後結果。

在 **updatefec()** 時，另外也會記錄一個 **gate** 是否為 **fecgroup** 的開頭(**_fehead**)，以及在 **fecgroup** 當中哪一行，方便 **printing** 與之後判斷使用。

c. Randomsim() 終止條件評估:

在 **randomsim** 當中，若連續超過 700 次 **fecgroup** 的 **size()** 變化都在 **+/-5** 以內，就會終止 **simulation**。

若以 **sim13.aag** 跑 **simiulate -r**，改變終止次數結果如下(只跑一次 **command**)：

連續超過 x 次 fecgroup 的 size() 變化都在+-5 以內	Fecgroup 的 size()	Total pattern 數
x=100	3870	16192
x=200	3769	25344
x=300	3681	32832
x=400	3570	40192
x=500	3479	48000
x=600	3442	54976
x=700	3403	61824
x=1000	3295	82176

基本上 x=1000 時產生的結果與 reference 最相近，但 simulate 的時間需 10.4 秒，且組數較少可能一組裡面更多 group，不一定比較好，故最後我選擇 700 次的當做最終版本，因其只需 8.27 秒完成，且觀察組數，可見其下降到快下不去時即結束，較無 overhead。

d. 附屬 command:

甲、 Printfecpair()

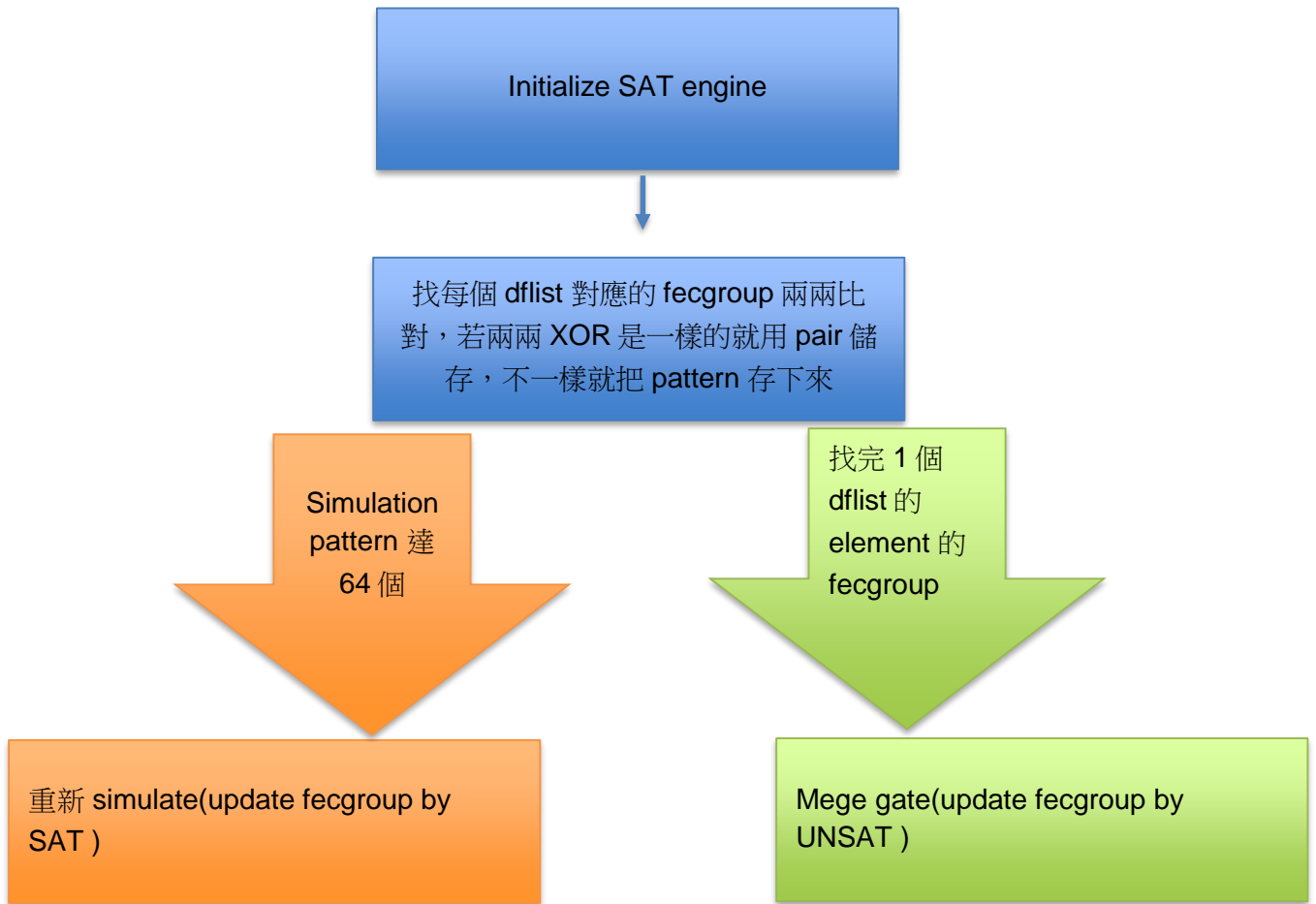
因須按照 id 順序來 print，故在 updatefec() 中我會在 fecgroup 每一排的開頭將 _fechead 設為 true，而當叫到這行 command，就會從 id=0 往下找，若在 fecgroup 中且為 _fechead 時就會輸出那一排全部的 gate，如此一來就只需要 print 一排的時間複雜度 $O(n)$ 而已

乙、 ReportGate():

Report gate 中還要在 fecgroup 中找出對應的 gate，但因有存取 gate 在 fecgroup 中哪一行，找到對應行直接輸出全部行數就行了，也是 $O(n)$ 。

5. CirFraig:

我做到大致流程如下:



Sim01.aag 和 sim02.aag 都可跑出來，但較大的會有 segmentation fault，應該是一些 flag 沒有 maintain 好導致。不然邏輯上應該沒問題。