# 2018-1
# Data Structure and Programming
# Final Project Report

Student: 電機二 呂承樺 B06901011
Email: b06901011@ntu.edu.tw
Cellphone: 0911-490-452
Facebook: Anita Lu

# Table of Contents

# I. My implementations and my design of data structure

## A. My data members of classes

### 1. class CirMgr

```
ofstream                *_simLog;
bool                    strashed;
vector<CirGate*>        _piGate, _poGate, _AIG,  _totalGate, _dfs;
vector<FECGroup*>       _FECList;
vector<int>             spec;
```

(a) bool strashed

在電路剛讀進來時，初始值設為 false，一旦 strash 過後，設定成 true 避免不必要的重複 strash。

(b) vector<CirGate*> _piGate, _poGate, _AIG, _totalGate

四種不同的 list 用來儲存 gate。其中，_totalGate 用來儲存所有種類的 gate，包含 PI, PO, AIG, CONST, UNDEF, NON。主要方便透過 id 的形式呼叫特定的 gate。另外，_piGate, _poGate, _AIG 則分別用於儲存 type 為 PI, PO, AIG 的 gate，便於在 gate reporting 的時候不用 traverse 整個_totalGate list。

(c) vector<CirGate*> _dfs

這個 list 是在執行 CIRRead 時透過 depth-first traversal 所建出來的 list，便於找從 PO 開始可以找到的 gate。不用每次呼叫到的時候都要重新 recursive 一次整個電路。

(d) vector<FECGroup *> _FECList

這個 list 用來儲存整個電路中的 functionally equivalent candidate pairs，有關 FECGroup 的其他部分，會在 class FECGroup 說明。

(e) vector<int> spec

我定義了一個 vector<int>用來存 AIGER 格式檔，從 spec[0]到 spec[4]分別儲存 M(maximal variable index)、I(inputs)、L(latches)、O(outputs)、A(AND gates)，其功能為初步檢查電路是否合理，也同時可以檢查 L 在本 project 中設定為 0。

## 2. class CirGate

```
unsigned              _id, _lineNo, _gatetype;
bool                  invert0, invert1, isReported, isMarked, isInDFS, isInFECList;
int                   _FECid;
string                _name, _result, _totalResult, _value;
CirGate*              fanin[2];
vector<CirGate*>      fanout;
Var                   _var;
```

(a) unsigned _id

_id 是每個 gate 本身在 AIGER 格式檔中被宣告的 ID，PI 和 AIG 的 ID 是 literal 除以二，而 PO 的 ID 則是以 header 讀入的 M 值(spec[0])再按照順序往上疊加。

(b) unsigned _lineNo, _gatetype

_lineNo 是用來記錄在 AIGER 格式檔案被 parser 時，所宣告的每個 gate 對應的行號跟欄位，_lineNo 在 reportGate 這個回報 gate 相關資訊的 function 會使用到。_gatetype 則是用來判斷 gate 的型態所設的參數。

(c) bool invert0, invert1; CirGate* fanin[2]

fanin[2]是用來記錄每個 AIG 輸入端的兩個輸入 gate，而 bool 則是紀錄兩個 fanin 是否有 invert。

(d) vector< CirGate*> fanout

由於每個 gate 的輸出端能接到的 gate 沒有數量限制，所以我把每個 gate 的 fanout 儲存成一個 list，方便要呼叫時不用再 traverse 一次整個電路。

(e) bool isReported, isMarked

isReported 是用在 CIRGate 輸出 fanin 和 fanout 時，判斷是否輸出過，避免重複的電路不斷被印出。isMarked 則是用在 gate 是否被 traverse 過，還沒走過的設為 false，一旦走過便設成 true，防止同樣電路不斷被重複經過。

(f) bool isInDFS, isInFECList

判斷每個 gate 是否在_dfs 的 list 當中，每重建一次_dfs list 時就要重新判斷一次，以便於在 CIRSWeep 時清理。isInFECList 則是判斷 gate 是否在 FECList 裡面，在實作 fraig 的時候可以用來判斷是否對該 gate 進行 SAT 的證明。

(g) string _name

在 AIGER 格式檔中在 A 之後 c 之前的定義，用於定義 PI 和 PO 的名稱。

(h) string _result, _totalResult, _value

在判定 FECGroup 時，在每個 gate 中，每 64 種模型會被輸出成一組 result，並被加入_totalResult 中，而最後執行的一組 result 會透過轉換，成為印出來的_value。

(i) size_t _FECid

用來記錄 FECList 中每個 list 的 ID，在呼叫 FECGroup 的時候比較方便。

(j) Var _var

在實作 fraig 的時候用來儲存每個可以被 depth-first-search 到的 gate 在 CreateCNF 後產生的 var，其中每個 poGate 的 var 與其唯一相連的 AIG 相同，其餘每個 gate 都有獨一無二的 var。

(k) 繼承 classes

右圖四個 classes 都是繼承於 class CirGate，其 data member 與 class CirGate 完全相同，但是在使用上略有不同。

```
class PIGate : public CirGate{};
class POGate : public CirGate{};
class AIG : public CirGate{};
class CONST_0 : public CirGate{};
```

### 3. class FECGroup

```
size_t                                  id;
vector<pair<CirGate*,bool>>             FECG;
```

這個 class 的建構主要是為了存取 CIRSIMulate 所得到的 functionally equivalent groups(包含 Inversely functionally equivalent groups)，每個 FECgroup 是由一群在 CIRSIMulate 後結果極為相似的一群 gate 所組成，用以在 CIRFraig 時進行進一步分析。

(a) size_t id

記錄每個 group 的 id，有助於排序以及呼叫。

(b) vector<pair<CirGate*, bool>> FECG

此 vector 存取 group 中所存有的 gate，而 bool 則紀錄其與 group 中第一個 gate 是否為 functionally equivalent groups(否則為 Inversely functionally equivalent groups)。

## B. My member functions of classes

### 1. class CirMgr

(a) Member functions about circuit construction

```
bool readCircuit(const string&);
void addFECList(FECGroup* g){ _FECList.push_back(g); }
void DFS(CirGate* gate);
void resetMark();
```

(1) bool readCircuit：主要用在 CIRRead 初步讀檔。

(2) void addFECList：將 data 丟入_FECList 裡面。

(3) void DFS：recursive 的 depth-first search，用來建立_dfs list，方便之後用來搜尋 gate 的 fanin 和 fanout。

(4) void resetMark：在完成 DFS 後清除所有已經 mark 過的 gate，以備下次的 DFS。

(b) Member functions about circuit optimization

```
void sweep();
void optimize();
int  OptCases(CirGate*);
void CleanFanout(CirGate*,CirGate*);
void CutAndConnect(CirGate*,CirGate*,bool);
void Cout(CirGate*,CirGate*, bool);
```

(1) int OptCases、void CleanFanout、CutAndConnect、Cout：這四個 functions 都是用來輔助 sweep 和 optimize，而其中 CleanFanout 和 CutAndConnect 在 fraig 時也會用到。

(c) Member functions about simulation

```
void                randomSim();
void                fileSim(ifstream&);
void                setSimLog(ofstream *logFile) { _simLog = logFile; }
void                AIGSimCal(CirGate*);
void                POSimCal(CirGate*);
void                CreateFECList();
void                ResetFECList();
bool                Equal(CirGate*,CirGate*);
string              Invert(string);
string              ResultToValue(string);
string              GenerateInput();
vector<FECGroup*>   getFECList() { return _FECList; }
```

(1) void AIGSimCal、POSimCal：分別計算 AIG 和 PO 兩種 gate 的 simulation value。

(2) void CreateFECList：在每個 gate 都得到 simulation value 後，透過比對每個 AIG 的結果來區分成多個 FECGroup，並儲存至 _FECList 中。

(3) void ResetFECList：無論是在執行第一次 CreateFECList 或是在完成 CIRFraig 後都必須重新設定整個 FECList。

(4) bool Equal：用來判定兩個 gate 的 simulation value 是否完全相同或是完全相反。

(5) string Invert：用來將整個 simulation value 倒轉(0⇔1)。

(6) string ResultToValue：主要用在 reportGate 時需要輸出最後執行的一組 64 bit 的 simulation value，並在每 8 個 bit 中加入「_」方便閱讀。

(7) string GenerateInput：在 randomSim 中需要自行產生每次 64 bit 只含 0 和 1 的測試值，利用 c++內建亂數種子 mt19937_64 每次產生一組由 0 和 1 組成的 string 回傳。

(8) vector<FECGroup*> getFECList：用來呼叫特定的_FECList。

(d) Member functions about fraig

```
void strash();
void merge(CirGate*,CirGate*);
void fraig();
void CreateCNF(SatSolver*);
void FraigMerge(CirGate*, CirGate*, bool);
bool Prove(SatSolver*, CirGate*, FECGroup*);
```

(1) void merge：在 strash 中用來 merge 兩個 gate。

(2) void CreateCNF：在 fraig 中透過 SatSolver 來證明兩個 gate 等效，需要先把每個 gate 附上唯一的 var 值，並將每個 AIG 與其兩個 Fanin 連接起來。

(3) void FraigMerge：在 fraig 判斷完為 UNSAT 後會將兩個 gate merge 起來，並清除另一個 gate 的所有函式集合。

(4) void Prove：判斷被呼叫到的 gate 與它所在的_FECGroup 裡是否有 UNSAT 的 gate，有的話執行 FraigMerge，若為 SAT 則將被呼叫到的 gate 移除原本的_FECGroup。

(e) Member functions about circuit reporting

```
void printSummary() const;
void printNetlist() const;
void printPIs() const;
void printPOs() const;
void printFloatGates() const;
void printFECPairs() const;
void writeAag(ostream&) const;
void writeGate(ostream&, CirGate*) const;
```

(1) 以上所有 function 皆與 CIRPrint 有關。

## 2. class CirGate

(a) Basic access functions

```
unsigned getLineNo() const           { return _lineNo;       }
unsigned getID() const               { return _id;           }
bool getInvert0() const              { return invert0;       }
bool getInvert1() const              { return invert1;       }
bool getMark() const                 { return isMarked;      }
bool getIsInDFS() const              { return isInDFS;       }
int getFECid() const                 { return _FECid;        }
CirGate* getFanin0() const           { return fanin[0];      }
CirGate* getFanin1() const           { return fanin[1];      }
CirGate* getFanout(size_t i) const   { return fanout[i];     }
string getName() const               { return _name;         }
string getValue() const              { return _value;        }
string getResult() const             { return _result;       }
vector<CirGate*> getFanoutList()     { return fanout;        }
size_t getFanoutSize() const         { return fanout.size(); }
Var getVar() const                   { return _var;          }
```

(b) Basic set functions

```
void setFanin0(CirGate* tmp)         { fanin[0] = tmp;   }
void setFanin1(CirGate* tmp)         { fanin[1] = tmp;   }
void setInvert0(bool b)              { invert0 = b;      }
void setInvert1(bool b)              { invert1 = b;      }
void setName(string tmp)             { _name = tmp;      }
void setresult(string str)           { _result = str;    }
void setMark(bool b)                 { isMarked = b;     }
void setReport(bool b)               { isReported = b;   }
void setIsInDFS(bool b)              { isInDFS = b;      }
void setType(GateType g)             { _gatetype = g;    }
void setFECid(int i)                 { _FECid = i;       }
void setValue(string str)            { _value = str;     }
void setVar(const Var &v)            { _var = v;         }
```

(c) Printing Functions

```
virtual void printGate() const {}
void reportGate() const;
void reportFanin(int level) const;
void reportFanout(int level) const;
void Fanin(CirGate* gate, int level, int space, bool invert) const{}
void Fanout(CirGate* gate, int level, int space, bool invert) const{}
```

## 3. class FECGroup

(a) Basic access functions

```
vector<pair<CirGate*,bool>> getGroup()  { return FECG;                        }
CirGate* getGate(size_t b)              {...                                  }
bool getInvert(size_t b)                {...                                  }
size_t getFirstMemId()                  { return FECG.begin()->first->getID(); }
size_t getFECGroupSize()                { return FECG.size();                 }
size_t getFECGroupId()                  { return id;                          }
```

都是為了拿到各種關於 FECGroup 的資料而開的 functions

(b) Other Useful functions

```
void addFEC(CirGate* g, bool b)                  {...          }
void erase(size_t b)                             {...          }
void sortEachGroup()                             {...          }
void setGroup(vector<pair<CirGate*,bool>> tmp)   { FECG = tmp;  }
size_t setFECGroupId(size_t d)                   { id = d;      }
void printFECGroup()                             {...          }
string printOther(size_t id)                     {...          }
```

(1) void addFEC：用來將某個 gate 加入此 FEC Group 中的 function，傳入的 CirGate*和 bool 會被組成 pair 的形式存入，其中 bool 是用來判斷是否為 functionally equivalent(否則為 inversely functionally equivalent)。

(2) void erase：用來將某個 gate 從此 FEC Group 中移除的 function，用在整理 FECGroup 以及 fraig。

(3) void sortEachGroup：每個 FECGroup 在完成建立後，必須按照 gate 本身 id 大小排列所用的 function。

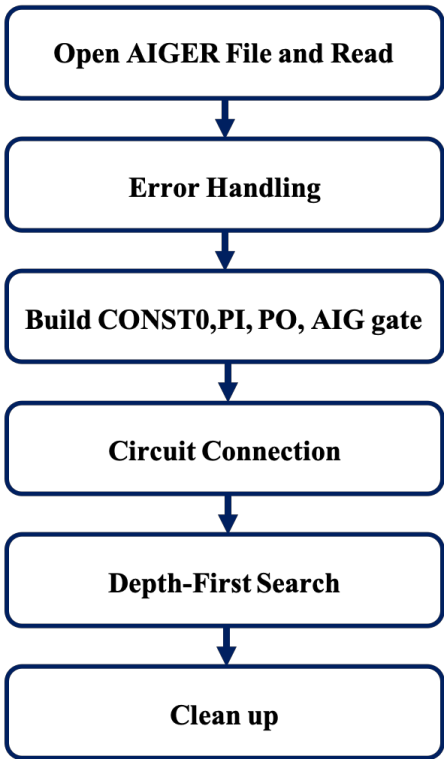
(4) void setGroup：在 sortFECGroup 時，交換 Group 位置時使用。

(5) size_t setFECGroupId：在 sortFECGroup 時因為 Group 位置的交換，連動必須跟著更改裡面每個 gate 的 FECid 所設的 function。

(6) void printFECGroup、string printOther：第一個用在 CIRPrint 中需要印出所有 FECGroup 以及其內的所有 gate。第二個則用在 reportGate 時，將除了本身之外的其他 gate 印出來。

# II. My implementations for commands supporting

## A. CIRRead：read in a circuit and construct the netlist parser




主要是用來讀進 AIGER 格式檔案的 command，首先先將檔案開啟並確定開啟的檔案是否正確。接著讀進檔案的第一行，也就是「MILOA」，並做基本的 error 處理，確定檔案格式與電路邏輯皆合理正確。

接著，便是透過 AIGER 格式檔案每一行的定義創建電路，其中我將_totalGate list 設定一個大小為 M+I+L 的 list，用以將每個 gate 直接透過 id 對應到隸屬的 _totalGate，所以若為沒有被定義到的 gate，其 gate type 便設定成「NON」，方便確認 gate 的存在性。在 PI、PO gate 都設定完成後，才進行 AIG 的創建，透過一個二維陣列先將每個 AIG 的兩個 fanin gate id 存起來，方便之後一起做 connection。等所有被定義到的 gate 都創建完成後，才進行 connection 的動作，避免在某些 gate 尚未創建就被呼叫到會造成 segmentation fault。

在整個電路完整 connect 後，便進行 Depth-first search 的 traversal 並將過程中一些不必要的資料整理乾淨或做清除。

## B. CIRPrint：print circuit

### 1. Summary

印出_piGate、_poGate、_AIG 的 list 大小，並計算總共的 gate 數量。其中，我使用 cout << right 來防止格式上的錯誤。

### 2. Netlist

按照 Depth-first search 的順序，從 PO 端開始印出所有往回可以連接到的 gate 並印出。其中分成 PI、PO、AIG、CONST 這四種形式討論，這個 command 主要是用在檢查其他 command 執行後電路是否依舊正確。

### 3. PI

按照_piGate list 的順序印出所有_piGate 的 id。

### 4. PO

按照_poGate list 的順序印出所有_poGate 的 id。

### 5. Floating Gate

這個部分我開了兩個 vector 分別用來儲存 with floating inputs 和 unused gate 的 id，在整個_totalGate list 中搜尋，只要遇到符合的 gate 就 push_back，整個 list 跑完後在一起印出來。

### 6. FECPairs

直接呼叫 class FECGroup 中的 printFECGroup function。

### 7. writeAag

將整個電路中有被 Depth-first search 到的 gate 按照 Depth-first search 的順序及格式印到一個指定的 aag 檔案中。

## C. CIRGate：report a gate

### 1. Gate

將該 gate 的所有相關訊息印出，包括 gate type、被定義在 AIGER 中的行數、是否有被定義名稱、以及是否有在 FECGroup 中，有的話透過 class FECGroup 中的 printOther 印出其他同在一個 group 中的 gate id。另外，若有完成 simulation，則印出最後執行的 64 bit simulation value。

### 2. Fanin

呼叫 class CirGate 中的 Fanin function，透過 recursive 的方式往 input 端去搜尋小於或是等於 level 的 fanin gate，且紀錄哪些 gate 是已經 report 過的要將 isReported 設定為 true，避免重複印出很多一樣的 gate。

### 3. Fanout

呼叫 class CirGate 中的 Fanout function，透過 recursive 的方式往 output 端去搜尋小於或是等於 level 的 fanout gate，且紀錄哪些 gate 是已經 report 過的要將 isReported 設定為 true，避免重複印出很多一樣的 gate。

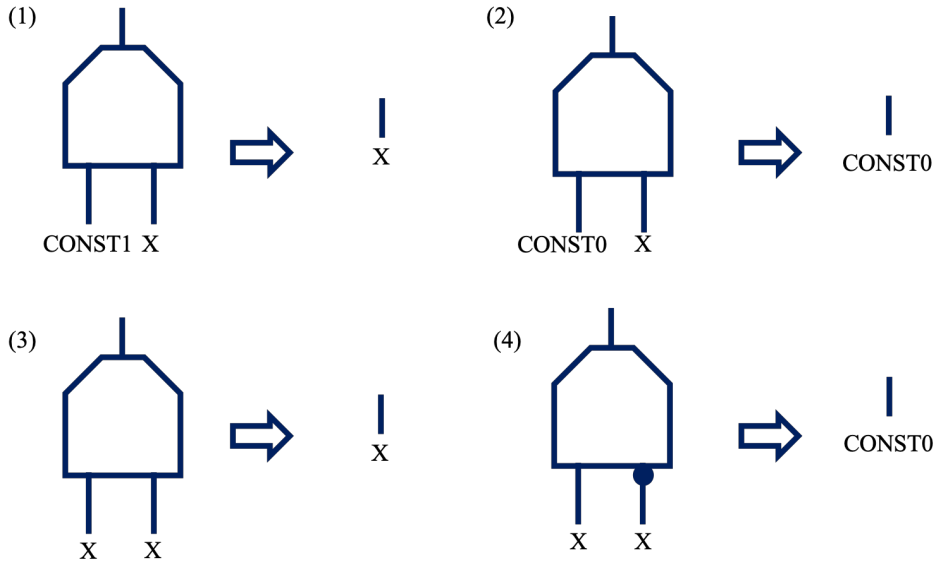
## D. CIRWrite：write the netlist to an ASCII AIG file (.aag)

主要就是用基本 cout 的方式將全部要輸出的東西串進 outfile 裡面，其中要注意的是 AIG 的輸出不是按照原本 AIGER 格式檔案中的順序，而是按照 depth-first search 的順序印出。

## E. CIRSWeep：remove unused gates

這個部分分成兩個區塊來執行。第一部分是將所有_AIG list 中不在_dfs list 中的 gate 都移除，並透過 CleanFanout function 將 gate 與其 fanout 斷開連結。第二部分則是搜尋整個_totalGate list 將 gate type 是 UNDEF 或是在第一部分中被移除的 gate 清理掉，並將 gate type 設成「NON」。

## F. CIROPTimize：perform trivial optimizations




為滿足 optimize 中的四種不同的情況，將每一個_dfs list 中為 AIG 的 gate 拿去跑 Optcases，若非以上四種情況則回傳 0，若滿足其中一種狀況則回傳 1~4 的值，並透過 CutAndConnect function 以及 CleanFanout 進行電路重整。其中要注意的是我並不會直接把要刪除的 AIG 直接從 list 中刪掉，而是先將該 gate 的 type 改成「NON」，這樣的好處是之後在一併清除 gate 的時候不會動到原本 list 中的順序，可以不用每次都重新找 gate 的位置。

而在清除完該清除的 gate 後要做的就是重新跑 depth-first search，確保_dfs list 的可用性。

## G. CIRSTRash：perform structural hash on the circuit netlist

```
class HashKey
{
 public:
      friend class CirGate;
    HashKey(size_t fan0, size_t fan1, bool in0, bool in1){
       if (fan0 > fan1){ f0 = fan1; f1 = fan0; invert0 = in1; invert1 = in0; }
       else if (fan0 == fan1){
          if (in0 > in1) {f0 = fan1; f1 = fan0; invert0 = in1; invert1 = in0; }
          else {f0 = fan0; f1 = fan1; invert0 = in0; invert1 = in1;}
       }
       else { f0 = fan0; f1 = fan1; invert0 = in0; invert1 = in1; }
    }
    ~HashKey(){}
    string operator() () const {
       string fan0 = to_string(f0), fan1 = to_string(f1), in0 = to_string(invert0), in1 = to_string(invert1);
       string key = fan0 + "+" + fan1 + in0 + in1;
       return key;
    }
    bool operator == (const HashKey& k) const { return false;}
 private:
   size_t       f0;
   size_t       f1;
   bool         invert0;
   bool         invert1;
};
```

這個部分我是透過 unordered map 來實作。而 unordered map 的 key 是透過上圖的 class HashKey 產生，我將每個_dfs list 中的 AIG 拿去跑 HashKey，方法是將該 gate 的兩個 fanin 以及兩個是否有 invert 傳入，並將兩個 fanin 小的 id 在前，大的在後，產生一個 string(例如:若 fanin id 是 100 和 200，且 invert 分別為 true 和 false，則生成的 key 為「100+20010」。)兩個 id 中間透過一個加號區分開，避免 key 的混淆。得到 gate 的專屬 key 之後，去搜尋 map 裡面是否相同的 key，若沒有則表示此為獨一無二不可取代的 gate，若有，表示該 gate 可以被 map 中同 key 的 gate merge 掉，並執行 merge 和 CleanFanout，並將該 gate 的 type 設成「NON」。簡言之，執行完整個 strash 後，每個 map 中一個 key 只會對應到一個 gate，其餘的都必須 merge。

完成 strash 後必須重新整理每個 gate 的 type，並重新進行 depth-first search，確保_dfs list 的可用性。

## H. CIRSIMulate：perform Boolean logic simulation on the circuit

在 III .My algorithm of Simulation 說明。

## I. CIRFraig：perform Fraig operation on the circuit

在 IV. My algorithm of Fraig 說明。

# III. My algorithm of Simulation

## A. fileSim

這個部分是透過讀進檔案來執行。我的實作方法導致 Simulation 的執行時間明顯偏長，會在後面做說明。我先透過 getline 將檔案中的每一行讀進來，並建立了三個 string 陣列用來存取。pattern 的陣列大小與_pi list 相同，用來存取每個_piGate 的輸入 simulation value。接著每次將長度為 64 bit 的 pattern 輸入為_piGate 的_result，並透過 AIGSimCal 和 POSimCal 來將每個 gate 的結果都存進_result 中。接著呼叫 CreateFECList 來建立 FECList，若_FECList 是空的，則創建，否則便是將每個 FECGroup 中的 gate 拿去跑 Equal，若結果為 true 則留下，結果為 false 則將該 gate 存進名為 rest 的 list 中再去做一次 CreateFECList。結束 CreateFECList 後，進行 SortFECGroup 確保_FECList 中的順序正確，以便下一次的呼叫。每完成一組 64 bit 的 FECList 重整後必需將每個 gate 的_result 清空，以便進行下一組計算。所有計算跑完後則透過 ResultToValue 來產生在 reportGate 中要印出的 value。

## B. randomSim

這個部分是要自行隨機產生 pattern 放進電路中執行，所以透過 GenerateInput 產生該電路 PI 數量組的 input 來執行，其餘的執行步驟皆與 fileSim 類似，在此不贅述。

這個 function 比較特別的點是我們必須自行判斷要執行多少次 simulation，因為執行次數會導致 FECGroup 是否分得夠細緻，使得在執行 CIRFraig 的時候不用做太多不必要的證明。這個部分將在 V. Results and analysis of Experiments 的地方做說明。

# IV. My algorithm of Fraig

這個 command 主要是將在 CIRSIMulate 中產生的_FECList 中的每個 gate 去證明是否真的 equivalent。

首先，為了要讓在 CIRSIMulate 中產生的 functionally equivalent groups 能夠被 Boolean Satisfiability Engine 這個系統 proof 是否真的為 equivalent，因此第一步必須先做出一個 SatSolver 的 object，我取名為 ss，接著 initialize 這個 object。第二步則是要建立 CNF。所以呼叫 CreateCNF function。此 function 先將 CONST0 的 var 設成 0，接著將每個 PI、AIG 透過呼叫 newVar 產生獨一無二的_var，並存入 map 中以利存取。接著，將每個 PO gate 的_var 設定其 fanin 的 AIG 的_var，最後再呼叫 addAigCNF 將每個 gate 連起來，完成 CNF 的建立。

整個電路完整建立後，便透過_dfs list 的順序來執行證明，只要該 gate 的 isInFECList 是 true 且 type 沒有在過程中被設成「NON」，並執行 Prove 的過程。

在執行 Prove 的過程中，我分成兩種 case，第一種是該 FECGroup 中有 0 的 case，另一個則否。若該 FECGroup 中有 0，且該 gate 與 0 在 CIRSIMulate 後產生的_result 是相同的，則證明該 gate 是否等於 1，若相反則證明是否等於 0。透過 assumpSolve 來得到證明結果，如果結果為 SAT 則回傳 1，代表兩個 gate 不相等，若結果為 UNSAT 則回傳 0，表示兩個 gate 等效，便透過呼叫 FraigMerge 來把該 gate 用 0 merge 掉。第二種情況為一般情形，將該 gate 與同 Group 中的其他 gate 進行證明，若結果為 SAT 則將該 gate 從 FECGroup 中移除，避免不必要的重複證明，若結果為 UNSAT，則將相比的 gate 用原本的 merge 掉，並將它從 Group 中移除。

在完成全部證明後，便開始重新整了所有的資料。包括將_AIG list 中 type 變為 NON 的 gate 清除，重新做 depth-first search 確保_dfs list 的可用性。

# V. Results and analysis of Experiments

## A. About MAX_FAIL in randomSim

由於 randomSim 需要自己判定不同電路需要跑的 pattern 數量，以求在執行 CIRSIMulate 和 CIRFraig 的時間可以達到最短，所以必須對 MAX_FAIL 進行進一步分析。

由於我的程式在執行 CIRSIMulate 過程中使用 string 作為 pattern 的 data type，且用 vector 來儲存 FECList，所以我假設電路的 PI 數目越多，在 simulation 的時間就會越長，所以為了平衡執行時間，我會盡可能讓我的程式不要執行過久的的 simulation，也就是希望 MAX_FAIL 越小越好，能夠在跑最少次數完成大致的 FECList 分類即可進入 Fraig 做分析。下表是我做的實驗。

sim06 (PI = 4, AIG = 4270)

| MAX_FAIL | 1 | 2 | 3 | 4 | 5 | 10 | 20 | 30 | 40 |
|---|---|---|---|---|---|---|---|---|---|
| AIG after fraig | 2625 | 2625 | 2625 | 2625 | 2625 | 2625 | 2625 | 2625 | 2625 |
| time(s) | 3.7 | 3.72 | 3.68 | 3.62 | 3.93 | 3.89 | 4.11 | 4.24 | 4.12 |
| memory(M) | 5.894 | 5.883 | 5.91 | 5.961 | 5.832 | 5.789 | 5.918 | 5.992 | 5.875 |

sim07 (PI = 4, AIG =9473)

| MAX_FAIL | 1 | 2 | 3 | 4 | 5 | 10 | 20 | 30 | 40 |
|---|---|---|---|---|---|---|---|---|---|
| AIG after fraig | 2658 | 2135 | 2135 | 2135 | 2135 | 2135 | 2135 | 2135 | 2135 |
| time(s) | 65.47 | 53.83 | 54.11 | 55.99 | 57.37 | 56.47 | 58.06 | 62.61 | 62.3 |
| memory(M) | 13.57 | 13.29 | 13.28 | 13.21 | 13.36 | 13.24 | 13.16 | 13.56 | 13.18 |

sim09 (PI = 178, AIG =3286

| MAX_FAIL | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 20 | 30 | 40 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| AIG after fraig | 1312 | 1300 | 1290 | 1299 | 1290 | 1281 | 1284 | 1283 | 1278 | 1283 | 1275 |
| time(s) | 4.96 | 4.41 | 4.46 | 4.86 | 4.99 | 4.56 | 5.17 | 5.32 | 5.72 | 6.43 | 7.5 |
| memory(M) | 4.695 | 4.645 | 4.605 | 4.73 | 4.602 | 4.797 | 4.82 | 4.727 | 4.742 | 4.773 | 4.742 |

sim10 (PI = 36, AIG = 716)

| MAX_FAIL | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 20 | 30 | 40 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| AIG after fraig | 308 | 308 | 307 | 295 | 291 | 287 | 285 | 285 | 285 | 285 | 285 |
| time(s) | 0.37 | 0.38 | 0.42 | 0.41 | 0.41 | 0.44 | 0.44 | 0.48 | 0.65 | 0.83 | 1 |
| memory(M) | 1.355 | 1.293 | 1.375 | 1.328 | 1.422 | 1.375 | 1.324 | 1.309 | 1.359 | 1.289 | 1.371 |

sim12 (PI = 277, AIG = 9463)

| MAX_FAIL | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 20 | 30 | 40 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| AIG after fraig | 5907 | 5920 | 5863 | 5857 | 5817 | 5849 | 5857 | 5846 | 5832 | 5796 | 5823 |
| time(s) | 59.26 | 60.42 | 61.38 | 57.81 | 55.34 | 56.2 | 60.66 | 56.37 | 58.62 | 62.97 | 65.09 |
| memory(M) | 13.32 | 13.02 | 12.54 | 14.38 | 13.91 | 12.82 | 13.42 | 13.93 | 13.72 | 14.32 | 14.67 |

sim14 (PI = 41, AIG = 886)

| MAX_FAIL | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 20 | 30 | 40 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| AIG after fraig | 639 | 636 | 630 | 628 | 620 | 621 | 617 | 616 | 614 | 611 | 611 |
| time(s) | 0.71 | 0.74 | 0.75 | 0.75 | 0.75 | 0.78 | 0.78 | 0.79 | 0.98 | 1.16 | 1.37 |
| memory(M) | 1.301 | 1.301 | 1.348 | 1.328 | 1.402 | 1.367 | 1.297 | 1.266 | 1.324 | 1.336 | 1.34 |

根據實驗結果，我發現其實影響執行的影響執行速度的不只是 PI 數目，當一個電路的 AIG 數目太多的時候，執行 CIRSIMulate 和 CIRFraig 的時間也明顯偏長。但由觀察發現若一個電路的 PI 數量很少，如 sim06 和 sim07，MAX_FAIL 只要大約 2 次就可以幾乎完全達到縮減電路的功能；而當 PI 數量為大約 50 個左右，如 sim10 和 sim14，則大約 MAX_FAIL=10 會是最佳效果；而當 PI 數量偏大，則 MAX_FAIL 越大，縮減電路效果越好，但若要平衡時間，取捨在 MAX_FAIL=20。

## B. About Fraig

我的 CIRFraig 雖然執行的還可以，但是當電路檔案很大，尤其是 AIG 很多的時候，因為是整個 FECList 的 gate 都證明，所以其實也需要跑好長一段時間，但是若把程式改成在一定程度證明後，先重整整個 FECList，再繼續執行 Fraig，我想執行時間應該會縮短許多。

## C. About Simulation

我的 Simulation 真的在架構上出了一些問題，若電路檔案過大(例如：sim13)，會在整個_dfs list 中來回跑好幾趟，增加很多時間(附圖為 sim13 執行 sim -f 以及 fraig 後的時間)，檢討後發現若是將 FECList 的建立用 unordered-map 及 Hash 來執行，而 simulation value 用 size_t 的 data type 來儲存，想必在執行速度上應該會提升不少。

my sim13

```
fraig> usage
Period time used : 3966 seconds
Total time used  : 3966 seconds
Total memory used: 164.5 M Bytes
```

ref sim13

```
fraig> usage
Period time used : 70.18 seconds
Total time used  : 70.18 seconds
Total memory used: 45.17 M Bytes
```

cirmiter result