# DSnP Final Project

*Functionally Reduced And-Inverter Graph*

*(FRAIG)*


系級：資管三

姓名：唐瑋廷

學號：b05705043

Email: b05705043@ntu.edu.tw

手機：0975655027

# 目錄

一、簡介 3

二、資料結構 3

*1. CirGate*

*2. CirMgr*

三、演算法 5

*1. Optimization*

*2. Strash*

*3. Simulation*

*4. Fraig*

四、綜合比較 10

五、心得 12

# 一、簡介

這次 project 主要是要處理 AIG 電路的簡化，分別用 sweep, optimization, structural hash, fraig 對電路進行檢測，進而去除一些不必要的 gate 。而整個程式部分，主要分為兩個 Class ，分別是 CirGate 和 CirMgr 。CirGate 是針對各種 gate 的實作，處理一些 gate 層級上的實作 (e.g. fanin & fanout)；CirMgr 則是對整個電路操作的實作，包含基本操作 (e.g. read, print...) 及簡化操作 (optimize, fraig…)。

# 二、資料結構

## 1. CirGate

**member variable:**

_id , _line: gate 的 id 和在 aag file 的 line number。

_faninPtr, _fanoutPtr: gate 的 fanin fanout 的指標，型態為 vector<CirGate *>。

_fanin, _fanout: gate 的 fanin fanout 的 id，型態為 vector<unsigned>。

_invert: 紀錄 fanin 是否反轉，型態為 vector<bool>。

_isVisited, _isReported: 分別用於標記此 gate 是否被 dfs 、gate report 過，於每次 dfs 及 report 清空。

_isFraiged: 用於 fraig 時，紀錄此 gate 是否已經被演算法遍歷。

_inDfs: 標記 gate 是否在 dfs list ，與前者的不同是前者會用在各種 traversal 場合，後者只會在建 dfs list 時更新。

_simValue: gate 最後一次 simulation 的 pattern，型態為 size_t。

_var: 用於 SAT solver，型態為 Var。

_fecGrp: 自己所在的 fec group 的指標，型態為 vector<CirGate *>*

**methed:**

getType(): 可以得到 gate type 的 enum 值。

getTypeStr(): 可以得到 gate type 的字串。

simulate(): 只有針對 PO, AIG 實作，PO 直接把自己的 _sim value 更新成 fanin 的；AIG 則會取兩個 fanin 的 sim value 做 AND。

註* 有實作 event-driven 和 all gate 兩個版本，但後來時做選擇只用 all gate，因此只用到其中一部份。

reportGate(): 印出該 gate 的詳細資訊 (e.g. id, symbol, sim value, fec group…)。

reportFanin(): 會以遞迴的方式去呼叫 fanin gates 的 reportFanin()，並印出自己的資訊。

reportFanout(): 同上。

2. CirMgr

**member variable:**

struct hearder: 存讀檔的 header 資訊 (e.g. pi number…)。

_gateList: 此電路所有的 gate，型態為 GateList (vector<CirGate *>)。

_fecGrps: 整個電路的 fec group list，型態為vector<GateList *>。

_PIOrder: 紀錄 PI 在 aag 檔宣告的順序，型態為 vector<unsigned>。

**method:**

buildDfs(): Dfs 整個電路，並且建立 dfs list。

change(del, replace, inv): 把 gate del 替換成 gate replace。

sweep(): 刪除不在 dfs list 的 gate。

optimize(): 把一些可以推理出 output 的 gate 取代。

strash(): 把 fanin 組成相同的 gate 合併。

randomSim(): 隨機產生 64 bits 的 sim pattern，assign 給所有 PI，然後 call simulate()。

fileSim(): 從 sim file 取得 sim pattern，assign 給所有 PI，然後 call simulate()。

simulate(): 對所有 gates 做 simulate，切割並收集 fec groups。

setFecGrp(): 在 simulate 完之後，把每個 fec group 存到對應的 gate 裡面。

fraig(): 對所有的 fec pair 做 SAT proving。

setVar(): 給所有 gate 一個 sat var ID。

constProofModel(): 建立每個 gate 的 CNF。

proveSat(): 對給定的兩個 gate 做 sat proving。

# 三、演算法

這個部分會針對以下四個函式作介紹，sweep因比較單純，因此不列入討論。

## 1. Optimation

```
for each AIG 'a' in dis list:
    if one of fanin of a is const 1:
        replace gate a with the other fanin
    else if one of fanin of a is const 0:
        replace gate a with const 0
    else if two fanins are identical:
        replace gate a with each of the fanin
    else if two fanins are invert:
        replace gate a with const 0
```

update dfs list

可以看到演算法的部分，主要是跑一次 dfs list，針對四種 case 做檢測及替換，並在演算法的最後做一次 dfs list 的更新。

## 2. Strash

```
HashMap hash
for each AIG ‘a’ in dfs list
        k = getKey(a’s fanins)
        if hash.check(k, oldGate):
                replace a with old gate
        else:
                hash.forceinsert(k, a)
update dfs list
```

這個演算法主要是跑一遍 dfs list，針對每個 AIG 的兩個 fanins，利用 hashMap 產生 key 的性質，以 O(1) 的方式去判斷這個 fanins 組合是否存在。

## 3. Simulation

**random simulate:**

```
while fail < max fail:
        for each PI:
                generate a 64 bits simulation pattern and set to sim value
        simulate()
        if fec group size not change: ++fail
        else fail = 0
sort fec groups list
set fec group list to every gate in the list
```

其中 max fail 是由 dfs list size決定：

if size < 100: max fail = 3

else: ln(size) / ln(5)

**file simulate:**

```
while input:
        collect sim pattern of each PI
        if already collected 64 patterns:
                simulate()
        if fec group size not change: ++fail
        else:                                   fail = 0
if there are some patterns still not simulated: simulate()
sort fec groups list
set fec group list to every gate in the list
```

**simulate():**

```
if fec group size == 0:  push all AIG and CONST to fec groups
for each AIG and PO in dfs list:   gate->simulate()
for each fec group in fec groups list:
        HashMap newFecGrps
        for each fec group in fec groups list:
                HashKey k(gate->getSimValue)
                if hash.check(k, grp):
                        grp->push_back(gate)
                else:
                        grp = new GateList
                        grp->push_back(gate)
                        newFecGrps.forceinsert(k, grp)
delete old fec groups list
collect new fec groups list
```

simulation 的作法使用 all gate simulation，因為經過實測，event-driven的方法並不會省下太多時間，卻讓整個 code 的複雜度增加不少，因此後來捨棄 event-driven 只採用 all gate。

4. Fraig

**fraig():**

```
while fec groups list size != 0 && fail < 5
    solver.initialize()
    setVar(solver)
    constProofModel(solver)
    for each unfraiged AIG in dfs list: (gate)
        case include const 0:
            only prove gate and const 0
            if isSat:
                store the sim pattern
                if already collected 64 patterns: resimulate()
            else if not sat:
                collect this gate pair (to merge later)
        case not include const 0:
            for each unfraiged gate of gate's fec group: (gate')
                prove gate and gate'
                 if isSat:
                    store the sim pattern
                    if already collected 64 patterns: resimulate()
                else if not sat:
                    collect this gate pair (to merge later)
    for each merge pair: replace a with b
    update dfs list
    update fec groups list
    if there are some patterns still not simulated: simulate()
    if fec group size not change: ++fail
    else:                                fail = 0
```

### constProofModel():

```
for each AIG in dfs list:
        call solver.addAigCNF
```

### proofSat(a, b, inv):

```
case const 0:
        solver.assumeRelease()
        solver.assumeProperty(a->getVat(), inv)
        ret = solver.assumpSolve()
case no const 0:
        Var v = solver.newVar()
        solver.addXorCNF(v, a->getVar(), false, b->getVar(), inv)
        solver.assumeRelease()
        solver.assumeProperty(var, true)
        ret = solver.assumpSolve()
if not ret: b->setfraiged
return ret
```

整個演算法是由 dfs list 出發，再拿 gate 的 fec group 出來一一比較，當證明兩個 gate 相等，會搜集該 gate pair；證明兩者不相同時，標記此 gate (後者) 為 isFraiged，避免後面繼續和他證明。而當搜集到 64 個 pattern 時，會啟動 resimulate，並跳出迴圈，從 dfs list 的前面再跑一次。

演算法的結束條件有二：

- fec groups list 被證明完
- fail 超過五次

前者不難理解，而後者是避免無窮迴圈的機制，以免大幅度影響 performance。

# 四、綜合比較

### Optimization:

| usage | ref code | my code |
|---|---|---|
| Total time | 0 | 0 |
| Total memory | 0.2x MB | 0.2x MB |

optimization 是用 run.opt.all 所測，差距和 ref code 不大，應該是因為 optimization 的複雜度是 O(n)，所以表現差不多。

### Strash:

| usage | ref code | my code |
|---|---|---|
| Total time | 0 | 0 |
| Total memory | 0.2x MB | 0.2x MB |

strash 的情況和 optimization 雷同，故不再贅述。

### fileSim:

**sim12.aag**

| usage | ref code | my code |
|---|---|---|
| Total time | 0.45 s | 1.33 s |
| Total memory | 3.477 MB | 5.52 MB |

**sim13.aag**

| usage | ref code | my code |
|---|---|---|
| Total time | 2.71 s | 9.61 s |
| Total memory | 19.62 MB | 39.11 MB |

file simulation 的部分，使用了較大的兩個 sim aag file 做測試，可以看到花費時間大約是 ref code 的 3~4 倍，推測原因可能是因為在實作 simulate 的時候，collect fec group 的時候是用另外的 tmp vector 暫存，造成太多的搬移導致效率降低、容量提高。

### randomSim:

#### sim12.aag

| usage | ref code | my code |
|---|---|---|
| Total time | 0.46 s | 0.48 s |
| Total memory | 3.512 MB | 5.438 MB |

#### sim13.aag

| usage | ref code | my code |
|---|---|---|
| Total time | 6.92 s | 11.9 s |
| Total memory | 18.96 MB | 39.04 MB |

random simulation 的部分，一樣使用了較大的兩個 sim aag file 做測試，因為我用的方法是計算 fail times，而 max fail 是一個 log 函式，也就是會隨著電路大小緩慢成長。sim12.aag 的表現和 ref code 差不多，而 sim13.aag 則是明顯慢了一些，推測原因可能是後面效率變低，可是 size 都還是有微幅變動，導致一直卡在低效率的情況。

### fraig:

#### sim12.aag

| usage | ref code | my code |
|---|---|---|
| Total time | 2.23 s | 12.83 s |
| Total memory | 6.863 MB | 9.32 MB |

#### sim13.aag

| usage | ref code | my code |
|---|---|---|
| Total time | 75.25 s | 361.1 s |
| Total memory | 44.77 MB | 58.48 MB |

fraig 的部分，可以很明顯的看到 performance 比 ref code 慢了 4~6 倍，原因可能是因為在前一個步驟的 simulate，分出的 fec group 品質不夠好，間接導致後面的 fraig 時間花得比較久，加上 fraig 本身的複雜度比較高，導致時間因素被更放大。

# 五、心得

因為已經修完系上的資料結構，所以一開始修課其實是抱著要挑戰更進階的資料結構的心態。結果發現老師其實一開始教的東西很基礎、很詳細，很多 C++ 的細節都解釋的很清楚，幫助大家釐清很多基本觀念。

學期的中段，開始進入偏向記憶體管理、資料結構的部分，開始感受到自己原本的資結觀念其實都還不夠扎實，也從來沒有去實作過各種結構，這部分因為這堂課，自己覺得進步很多。

而從一開始的作業，到最後的fianl project，老師都有在 spec 強調寫 code 要想在腦袋裡想一遍，前幾次作業還感受不太到這樣做的意義，而到後面幾次作業，規模越來越大，模組化、架構就顯得越來越重要，學期初的 const, pointer 的概念也都確實用在作業上，若沒有自己想過，永遠都不會了解這些東西的重要性。現在自己在寫 code 前，都會仔細的想好整個架構，盡量讓自己的 code 是模組化的，方便自己日後修改，也訓練自己寫出來的 code 是可以給大家所使用！

總而言之，很慶幸當初選擇修了這門課，再這裡學到的絕對不只有 C++、資結，更多的是想法、習慣！另外，身為外系來選修的學生，我覺得這門課完全不會讓人隔絕在外，上課也都跟得上老師的步伐，讓人覺得很窩心。