DSnP Final Project
-FRAIG-

系級：電機三
姓名：彭懷樘
學號：B03901038
聯絡信箱：b03901038@ntu.edu.tw

一、概述：

整個程式設計用於讀取一個由 AIG 構成的電路，藉由運算來簡化邏輯閘，甚至優化整個電路。最主要的程式架構是基於兩個 Class：CirMgr Class 和 CirGate Class。後者將每個各別邏輯閘(包含主輸入及主輸出)視為一個物件，並內含與其連接的邏輯閘資訊、還有該元件本身的重要資訊；而整個電路資訊是建構在 CirMgr Class 裡面，除了讀取與寫出，程式更透過 CirMgr Class 來進行整個電路的優化。

二、完成情況：

- Sweep ..... (✓)
- Optimization ..... (✓)
- Strash ..... (✓)
- Simulation ..... .(✓)
- FRAIG ..... (✘)

三、實作架構：

在此段落，我將會將兩個 Class 進行簡短的內在成員(member function)介紹，並對五個主要功能進行各別分析。

✓ Class CirGate

為了程式的可讀性，我讓電路中的 PiGate、PoGate、ConstGate、UndefGate 以及 AigGate 都繼承於 BaseGate。同時用 vector<CirGate*>儲存每個邏輯閘的輸入及輸出，並將反相的資訊藏於各自的指標之中，同時還存放了：

Protected Members:

- **bool _visited:** 在 Depth First Seaech 中用以標記此閘是否已經遍歷。
- **size_t _simValue:** 存取在 Simulation 階段的模擬值。
- **size_t _fecGrp:** 用以識別在 FECGroups 的組別。
- **Var _var:** 由 SAT Model 所建立的 ID。

Functions:

- DFSConstruct: 在電路更動(簡化/優化)後，進行 DFS 探索來更新 DFS 電路情況。

- Virtual void printGate: 由於各種型態的邏輯閘在回報時有不同要求，透過重新實作此函式，並共同呼叫 printGateDetail 來正確的印出，每個邏輯閘的型態、ID 以及 Simulation 的值。

✓ Class CirMgr

Private Members:

- **GateList _gateVarList:** 按照 ID 順序，存放所有當下存在電路中的邏輯閘記憶體位置。
- **GateList _dfsList:** 按照 DFS 順序，存放所有當下存在電路中的邏輯閘記憶體位置。
- **IdList tmpIn, tmpOut:** 存取了當下階段的所有 PI、PO 的 ID(按照 AAG 順序)，透過_gateVarList 能迅速得到該邏輯閘的記憶體位置。
- **List<FECGroup> _fecGrps:** 存取所有 FEC Groups。

Private Functions:

- **DFSConstruct / UpdateDFSList:** 在電路更動(簡化/優化)後，進行 DFS 探索來更新 DFS 電路情況。

- **Void replace(CirGate* old, CirGate* new, bool isInv, string& str):** 這個函式可謂五大功能的核心，目的是將傳入的 new 邏輯閘，取代 old 邏輯閘的位置，並藉由 isInv 的布林值來設定接角的相位，而 str 則是為了顯示現在運作的功能(Strashing..., Fraig...)，相當的具有使用彈性。期大致運作順序如下：
  1. 將 new 的 Fanout 與原先 old 的 Fanout 連結。
  2. 將 old 的 Fanin 與 old 斷開。
  3. 將 old 的 Fanout 與其應連結的邏輯閘連接。
  4. 將 old 從_gateVarList 中抹除。

Five Primary Functions:

- **Sweep Function:**

  藉由遍歷整個_dfsList 更改_visited 的布林值，並掃描整個_gateVarList 中尚未走過的邏輯閘，藉由該邏輯閘以及各自 Fanin 的型態，將無用的邏輯閘掃除。

- **Optimize Function:**

  藉由遍歷整個_dfsList，檢查所有 AIG 邏輯閘的 Fanin：

A. Const 1 Fanin: 將此閘 replace 成另外一 Fanin。
B. Const 0 Fanin: 將此閘 replace 成 Const 0。
C. Identical Fanin: 將此閘 replace 成此 Fanin。
D. Inverted Fanin: 將此閘 replace 成 Const 0。

- **Strash Function:**

為了搜尋相同兩輸入的 AIG 邏輯閘，我使用 HashMap 來進行雜湊運算，以增加比對效率。

我用每個 AIG 邏輯閘的兩帶有是否反向信息的記憶體位置進行雜湊，並對兩者進行簡單平移及加號運算，希望能一定程度降低雜湊時碰撞的機率。

接著在遍歷_dfsList 時寫入 HashMap 中，碰到相同邏輯閘時就進行取代已完成操作。

- **Simulation Function:**

在此我特別實作了兩個 class，Class SimValue 為包裹 Simulation Value，Class FECGroup 則是用於存放 FEC。

- Class SimValue:

  包裹起來的訊息是為了用於每個 FECGroup 的雜湊運算，使得程式更具有可讀性。

- Class FECGroup:

  整個 class 其實是一個堆疊 CirGate 指標的 vector。為了在 Simulation 還有 FRAIG 過程中不斷尋找每個 Group 之中的值，並拿來驗證 SAT，因此選擇了能夠 Random access 的 STL 來操作，其內也實作能連同反相訊息寫入記憶體位置的函式 FECGroup& add(const Cirgate* gate, const bool& inv)，還有一些簡易的功能，使得 Simulation 操作更加便利。

Simulation 主要由兩個相近的主要函式組成：randomSim()和 fileSim()。而這兩者最主要差異僅在於：初始 PI 餵入值，以及總模擬次數。整個函式經歷了主要五大步驟：

A. 邏輯閘模擬初始化：按照 DFS 遞迴地去訪問每個邏輯閘，並要求得到其

輸出 pattern(size_t _simValue)，因此會需要 PI 的初始輸入，如果是 fileSim()，則參照 pattern file 的格式，將 bit 一個一個吃進來之後，一口氣餵給 PI 去模擬(void simEachGate(firstTime))，若是不足 32/64bit，則全部補上 0。若是要做 randomSim()，我則使用 rnGen()來做簡單的平移以及 or 運算，盡量使得模擬的 pattern 不要重複，接著一口氣一組一組餵給 PI 進行模擬。

B. FECGroups 初始分組：初始模擬的同時，將進行 FECGroups 的分組，藉由 HashMap 的 query 布林值，判斷此邏輯閘與各組是否為 FEC/IFEC，是則加入該組，若不屬於任何組別則將另創新 FECGroup。

C. FECGroups 串接：將所有的 FECGroup 進行 list 的鍵結，若是 Group 內僅有一個邏輯閘，則將此組捨棄(CollectValidGroups)。

D. 邏輯閘模擬：假設輸入 pattern 為 32/64 的整數倍(不只一組)，則會連續進行模擬(void simEachGate(NOTfirstTime))，接著再次進行 FECGroups 分組，並同樣做 FECGroups 串接，周而復始的維護 list<FECGROUP>。

E. 記錄檔輸出：若是有要求輸出 log file，則每模擬 32/64 patterns 時，將此段 pattern，依照 PI、PO 的順序寫入*_simlog 中。

關於 MaxFail：

在 randomSim()時，由於不像 fileSim()，Simulation 結束時間由固定 pattern 數量決定，而是由一個 MaxFail 機制結束，程式設計如下：假使 MaxFail = 3，則在 FECGroup 組數在模擬三組 pattern 保持不動時(i.e. 8, 4, 3, 3, 3)，便停止模擬及分組，開始輸出 log file(如果有要求)。很明顯地，一個電路所需要的 MaxFail 絕對不是固定常數，而應該要是 Size Dependent，原本想對 ref program 進行數據迴歸分析，來檢驗所需的 MaxFaile 公式，但卻注意到迴歸分析的限制，必須大概知道回歸模型的樣子，否則誤判級數，也無法推出逼近公式。因此，我藉由以前從 python、Matlab 寫 Auto Image Segmentation by SuperPixels 的經驗：一張圖要被正確地切割，得先大致了解影像的複雜度(離散變異)及重心位置，而使程式自動決定初始分割 SuperPixels 的數量。透過平常常使用的公式(雖然不能直接套用，但精神相仿)，並透過一次一次的修正、比對，大致決定這個 MagicNumber 為 $\alpha \log_2(\#PI + \#AIG)$，並且使用$\alpha = 4$這個默認值，這個 MaxFail 次數經過每組 64bit 的 pattern 運算，能使我的 FECGroup 分得更完整、妥當。特別注意到在電路極大時(我有故意做過 cirmiter sim13.aag 四次的極端 case)，$\alpha = 2$也是個不錯的值，能有效降低 Simulation 的整體時間。

- **Fraig Function:**

這是一個透過先前經 Simulation 建立的 FEC/IFEC Groups，以及 SAT 系統進行布林等值運算的函式，由於先前我們是透過有限個數的 pattern 來證明組中的邏輯閘為潛在布林等值(Potentially FEC)，而透過此函式能證明出同組中兩邏輯閘是否真的等值。

此函式的演算方法可粗分為三大步驟：

A. 建構證明模型(createCNF)：透過遍歷_dfsList，將所有的邏輯閘設立一個 Var 識別變數，接著特別讓 AIG 邏輯閘建構模型(addAigCNF)。

B. 證明等值(solve)：對尚未訪問的邏輯閘，將其與他同組的邏輯閘進行 addXorCNF，並透過 SAT 引擎驗證：

   I. 如果此組內第一個邏輯閘為 CONST 0，那整組可以被相當迅速的驗證完畢。好比此組內有個!G[8]，驗證 G[8]=0 如果 SAT，若為 UNSAT，則 G[8]=1。
   II. 若並非上述情況，則兩兩證明彼此的等值關係：兩邏輯閘若為 SAT，則可以記錄珍貴的 input pattern，累積直到 32/64 組時，再次進行 Simulation 並更新 FECGroups；若兩者為 UNSAT，則呼叫 replace 進行取代。

C. 直到 FECGroups 數量歸 0 時，進行 Strash()以及 DFSConstruct()。

這邊有個尚未實作解決的問題，sim12.aag 似乎在經過 Strash()、Optimization()等小功能處理後，進行 Simulation 以及 Fraig，會出現邏輯閘輸出接向自身輸入的無窮迴圈。會無窮迴圈的原因在於 Simulation 在訪問 _simValue 的時候，因為呼叫不到此邏輯閘的 DFS 上一層而出錯，簡單構想為，設立一個計數器偵測在同一邏輯閘逗留的次數，超過一定門檻就強制猜測 pattern，如此應能跳出迴圈。

四、效能(sim13.aag 單一指令(i.e. cirr、cirsw、ciropt...)結果**)**

	My Program	Ref Program
CIRR	0.02 secs 19.67 MB	0.03 secs 12.38 MB
CIRSW	0.00 secs 19.67 MB	0.00 secs 12.38 MB
CIROPT	0.01 secs 19.67 MB	0.01 secs 13.24 MB
CIRSTR	0.02 secs 22.81 MB	0.00 secs 16.07 MB
CIRSIM -F (13)	22912 patterns simulated 1.94 secs 25.73 MB	22912 patterns simulated 2.59 secs 16.41 MB
CIRSIM -R	MaxFail = 65 14016 patterns simulated 2.42 secs 25.73 MB	MaxFail = 181 5792 patterns simulated 0.5 secs 16.41 MB
CIRF	N/A N/A	74.63 secs 45.26 MB

從讀取電路進來時，我耗費的空間就較大，原因在於我當初在讀取時存的vector<string>較多，為了留給後面 function 使用之故。因此扣除這個影響，每個功能呼叫所多耗費的空間就差不多了，至於 Simulation 的空間主要是來自於我的 FECGroups，當初是採用 list<FECGroup>，原因是在於想說，每個 Group 之間並沒有相當的強調順序性(除了排序)，特別是不需要 Random Access；而每個 FECGroup Class 採用的 vector<CirGate*>則是為了方便存取任意一個邏輯閘，空間損耗主要來自於此。

至於時間上面，撇開不穩定的 Fraig 來看(不失敗的情況下為 70~120 secs)，在 SAT 證明上的演算需要避免重複驗證。基本上前面的功能都沒有太大的差異，由於我是使用一次餵 64bits 的 parallel pattern，因此能比 Ref 快上許多，如果能增快 SplitFECGroup 的速度，也就是在 add Group 上頭做修正，應該能在更快些。

瓶頸：

目前所能測出的最主要瓶頸有二：

A. tmpOut 維護：

由於在優化電路的過程，有可能會 merge 到 AIG 邏輯值是直接連到 PO 的，好比 strash05.aag 做 cirstrash()時，會將 AIG 6 與 AIG 7 進行合併，因此原本的 PO 12(9)、PO 14(10)就要更動為 PO 12(9)、PO 12(10)，而好巧不巧，我 tmpIn、tmpOut 使用 IdList 型態，而且因為 HW6 的遺毒，tmpOut 存的 ID 還是連向他的 AIG 的 variable ID，因此在維護上面只能透過 DFSConstruct 去做修正，頗為麻煩。

B. cirSim 輸出檔：

因為輸出檔特殊的格式要求，使得原本是(#PI+#PO) X (64)的內存 pattern，要轉成(64) X (#PI+#PO)的輸出 Log，而且要按照吐出(也就是餵入)的順序列印，但我當下實作時並沒有注意到順序剛好顛倒，使得後來是宣告一個靜態矩陣去裝下這個 pattern，然後再多此一舉地倒著輸出，一個既花空間、又耗時間的作法...，因此在 AAG 檔增大時，一旦需要輸出檔案，耗費時間會增加不少(sim13.aag 做 fileSim()並讀取 pattern.13 時，時間從 1.93 秒增加為 9.37 秒，但 Ref 僅從 2.56 秒增加為 7.88 秒)。