2019 DSnP Final Project

Functionally Reduced AND-INV Graphs

系級 ： 電機二
姓名 ： 范育瑋
學號 ： b07901047
email： b07901047@ntu.edu.tw

Data Structure

CirGate:

這個 class 是這次整份程式中的主體，是一個抽象類別，下面繼承了 PIGate、POGate、AIGate、CONST、UNDEF，代表著五種不同的 Gate。

它主要的 data member 有

- id : aag file 裡對應的 id
- linNo : aag file 裡出現的列數
- simValue : simulation 時的 output
- faniList : 一個 vector<GateV>，用來存輸入以及它的相位
- fanoList: 一個 vector<GateV>，用來存輸出以及 gate 之間的相位

至於 member function 則只列出這次 final project 用最多的幾個 function

- deleteFano(CirGate* gate):
  找出 fanoList 中 argument gate 的位置，並將它移出 fanoList。

- replaceFani(CirGate* g, GateV gV):
  將 faniList 裡所有的 g 都換成 gV 裡存的 gate。

- pushFano(GateV fano):
  將 fano 這個 GateV 加進 fanoList 裡。

- replaceGate(GateV gV):
  將這個 gate 用 gV 來取代，流程為：對所有 fano, call replaceFani(this, gV)，gV call pushFano(fano)，最後再對所有的 fani call deleteFano(this)，就完成一次 merge Gate 的動作。

- Sweep():
  當這個 gate 要被 sweep 時呼叫，它會將所有 fano 跟自己 disconnect，並 delete this。

- Optimize():
  做 trivial optimization，如 const propogation，把 fani 之間的線根據 replace 的關係接好。

GateV:

作為一個 CirGate*的 wrapper class，利用 CirGate*這個 ptr 的 LSB 當 sign bit，因此它唯一的 Data member 就是一個 size_t。以下是幾個 member function

- gate():
  將 size_t 的 LSB 清掉轉成 CirGate* 回傳。

- isInv():
  與 0x1 做 bitwise operation，回傳 LSB 是否為 1。

- Operator bool ():
  為了程式碼的可讀性 overload 的 function。回傳這個 GateV 存的是否為 CONST0 && !isInv()。

- Operator ~ ():
  將 size_t 的 LSB flip 一次，並回傳一個新的 GateV。

- Operator == ():
  回傳兩個 GateV 的 size_t 是否相等。

- # CirSweep

  這個指令是將所有 unused gate 都刪除，因為只要不在 dfsList 裡，最後都會變成 unsused，因此我的做法是將所有不在 dfsList 裡的 gate 找出來，除了 PI 只需要把 fano 清空外，全部刪掉。

- # CirOpt

  我的作法是直接跑過一次 dfsList，並且去呼叫 CirGate::optimize()，簡化電路，其中我在 CirGate 的 static data member 裡存了一個 isNetChanged 的 flag，用來判斷 dfsList 是否有被更動，這樣一來若沒有可以 optimize 的 gate，dfsList 便不需要被 reconstructed。
  以下為參考 psuedo code。

  - CirMgr::optimize()

```
CirMgr::optimize(){
    for gate in _dfsList:
        if(gate is AIG)
            gate->optimize()
}
```

  - CirGate::optimize()

```
CirGate::optimize(){
    if(fanin1 == fanin2)
        this->replaceGate(fanin1)
    else if(fanin1 == !fanin2)
        this->replaceGate(const0)
    else if( !fanin1 || !fanin2)
        this->replaceGate(const0)
    else if( fanin1 || fanin2)
        this->replaceGate(fanin1?fanin1:fanin2)
}
```

- # CirStrash

這個指令是給定一組 gate pair，若他們有一樣的 Fanin 組合，則這兩個 gate 可以被 merge 成一個 gate。我的實做方法很簡單，為了節省記憶體，我選擇使用 set 而非 map，set 裡的 element 存的是 CirGate*，並提供一個自定義的 hash function object Strash，它 overload 了 operator()(CirGate* g)，會將這個 CirGate 的 兩個 fanin 的 unsigned Literal 以 32bit+32bit 的方式串接成一個 size_t，其中為了避免 fanin 排列的問題，一律以 id 由小到大接成一個 hash_key。

有了這個 hash function 後，接下來只要將 dfsList 裡的 gate 都丟進 set 裡 check，若存在則用 set 裡的 gate merge 外面的 gate；反之則將這個 gate 塞進 set 裡。最後一樣判斷是否需要 reconstruct dfsList，就完成了。
以下為參考 pseudo code。

```
CirMgr::strash(){
    Hash strash;
    for gate in dfsList:
        pair = strash.emplace(gate); // pair = <iterator, bool(isExist)>
        if(pair.second)
            pair.first.gate->replaceGate(gate);
}
```

- # CirSim

CirSim 這個指令主要分成兩部份：simulation 以及 FEC，首先來說明 Simulation 的實做。

- Simulation :
  由於實做的方便以及速度的考量，我選擇的是 64-bit 的 parrallel simulation，原先我想節省記憶體，以 32-bit 實做，但發現速度實在差太多，才改成與 ref code 相同的 64-bit。由於 dfs 的性質，我們能夠保證對於每個 PO，若根據 dfs order 走過，他的 fanin 一定也被走過，因此只要先對所有的 PI 輸入一組 simulation pattern，走訪過 dfsList 就完成了。
  至於如何控制 sim 的停止條件，我以最直接的方法，若連續 50 個 pattern fec 數量都沒有變，就會停止。這樣的壞處是會有一個 threshold pattern，不管有沒有 fecGrp 都至少要跑 50 次 pattern，好處就是 code 簡單粗暴。

- FEC:
  資料結構上，我在 CirMgr 存了一個 vector<IdList> _fecGrps，而每個 Gate 存的是在第幾個 _fecGrp。實做上我用了一個 map<fecKey, IdList> 來作為配對的容器，和 Strash 相當類似。

  - InitFEC():
    這個 function 會將所有的 AIGate 以及 CONST 0 放進同個_fecGrps 裡，此時_fecGrps.size()=1。

  - UpdataFEC():
    更新_fecGrps。對於_fecGrps 裡的每個 grp，我開一個 map<fecKey, IdList>來存分出來的更小的 grp，若 fecKey 不存在，則插入一個 pair<fecKey, IdList(thisLit)>的元素；反之則把 thisLit 推入 IdList。

需要注意的是，為了將 FEC 與 IFEC 都放在一起，這個 fecKey 會將 simValue 的 LSB 對齊成 1，若不是 1 的，則這個 gate 的 phase 就是 negative，lit = 2*id+1。

最後走過一遍這個 map，將 size>1 的存到 newFecGrps，把它 assign 給_fecGrps，就完成了。

- # CirFraig

我並沒有完成這個指令，因此僅就寫出的屍體以及想法做說明。Fraig 停止的條件為_fecGrps.size()==0，因此會有一個 while loop，每次迴圈都對於當前的 dfsList 做 solver 的 initialize()以及 assign var，接下來走訪過 dfsList，若存在 fecGrp，則對裡面的每個 gate 做 satProve，若 UNSAT 則用當前的 gate 作為 key 並把 grp 裡的 gate 存入 value 中，若是 SAT 則 collect 這個 parrellel pattern，證過一遍後把自己的 id 從這個 grp 裡刪除。最後根據再把先前存的 map，對於每個 key 把 value 裡所有的 gate 都用 key 來 merge，就完成了。

其中若 SAT pattern > 8 時，我會強制跳出來去用這個 pattern resimulate。

沒有實做出的原因主要是，最開始我沒有考慮 gate 之間 merge 的順序，若不是由 dfs 先碰到的 merge 後面的 gate，會出現從靠近 PO 的一側 merge 靠 PI 一側的狀況，導致 cycle 的產生，當我意識到這點時時間已經不夠了，很遺憾沒能完成最後的指令。

- # Experiment

由於 Sweep, Opt 演算法較為單純，因此以下只做 Strash 與 Simulation 的分析。

- Strash:
  使用的是 C7552.aag 左圖是我的，右圖是 ref code

```
fraig> usAGE
Period time used : 0.01 seconds
Total time used  : 0.02 seconds
Total memory used: 1.168 M Bytes
```

```
fraig> usAGE
Period time used : 0.02 seconds
Total time used  : 0.02 seconds
Total memory used: 0.875 M Bytes
```

可以發現在速度上大致差不多，而 memory 我用的比 ref code 多了不少，多的部份我想是差在我 CirGate 的 Data Structure 中，不論是哪種 gate，我都會先給它一個空的 faniList 與 fanoList，於是就多了很多 24*N bytes 的 overhead。

接下來為 sim13.aag 測試結果

```
fraig> cirr tests.fraig/sim13.aag

fraig> cirstrash

fraig> usAGE
Period time used : 0.07 seconds
Total time used  : 0.07 seconds
Total memory used: 21.23 M Bytes
```

```
fraig> cirr ../tests.fraig/sim13.aag

fraig> cirstrash

fraig> usAGE
Period time used : 0.08 seconds
Total time used  : 0.08 seconds
Total memory used: 17.15 M Bytes
```

這個 aag file 並沒有 merge 任何的 gate，速度依然差距不大，而 memory 則推測與前述的原因有關。

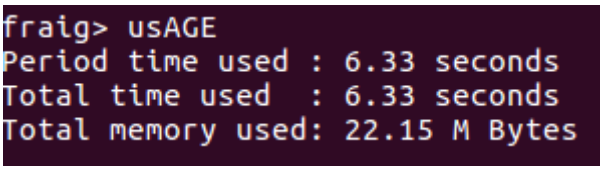
- Simulation:
  以下皆由 sim13.aag 作為測試檔。
  FileSim:

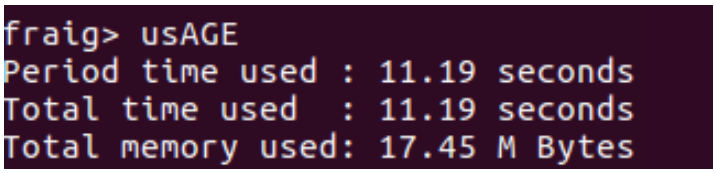
```
fraig> cirsiMulate -f tests.fraig/pattern.13
22912 patterns simulated.

fraig> usAGE
Period time used : 1.89 seconds
Total time used  : 1.89 seconds
Total memory used: 22.05 M Bytes
```

```
fraig> cirsiMulate -f ../tests.fraig/pattern.13
22912 patterns simulated.

fraig> usAGE
Period time used : 1.49 seconds
Total time used  : 1.49 seconds
Total memory used: 18.53 M Bytes
```

速度與記憶體依舊差距不大，其中我在實做 fileSim 的時候分別用了 Char array 與 string 來讀檔，發現前者比後者快了約 0.3 秒，而最終我的 code 是 string 的版本，因此推測速度為讀檔所造成的差距，接下來比較 randomSim。

RandomSim:





速度差異主要來自 simPattern 數量的不同，我的 code 總共跑了約 15w 組 64bit pattern，而 ref 跑了 30w 組，因此平均來說 sim 的速度依舊相同，另外比較兩邊分出來 fecGrp 的數量，我分出了 3268 組而 ref 分出了 3198 組，可以發現 sim 時間的代價換來的是接下來能夠節省的 sat solver 的 proof effort。