資料結構 Final Project

B03505052 工程科學及海洋工程學系四年級 李吉昌

Contact information：0972128853 / b03505052@ntu.edu.tw

HW6 架構使用：b04901132 電機三 林庭揚

Data Structure Design：

1. CirGate：

```
  string sim_value()
  {
    string str = sim_bits.to_string();
    vector<string> part;
    for (int index = 0; index < 64; index += 8)
    {
      part.push_back(str.substr(index, 8));
    }
    return (part[0] + "_" + part[1] + "_" + part[2] + "_" + part[3] + "_" + part[4] + "_" + part[5] + "_" + part[6] + "_" + part[7]);
  }

private:
  GateType _type; //the type of the gate
  unsigned line;
  int _ID;
  std::vector<unsigned> fanins;
  std::vector<unsigned> fanouts;
  std::vector<int> FECs;
  string symbol;
  bitset<64> sim_bits;
  void search_fanin(int, int, bool) const;
  void search_fanout(int, int, bool) const;
  void reset_visit() { is_visit = false; }
  mutable bool is_visit;
```

CirGate 的 class 設計中，變數如上，每個 Gate 中，fanins 及 fanouts 所儲存的值為 literal，存 literal 的原因是需要判斷輸入端對於該 gate 是否為 invert 或是該 gate 對輸出端是否為 invert 可以%2 是否為 0 來判斷。

sim_bits 則記錄每個 gate 在經過 simmulation 之後的結果，在 cirg 時使用 sim_value()將格式表達成 ref 的形式，而 FECs 則是記錄該 gate 屬於哪個 FEC pair(含 IFEC)。

2. CirMgr：

```
  void printSummary() const;
  void printNetlist() const;
  void printPIs() const;
  void printPOs() const;
  void printFloatGates() const;
  void printFECPairs();
  void writeAag(ostream &) const;
  void writeGate(ostream &, CirGate *);
  void connect(int, int); //connect the current gate with previous one
  void DFS(unsigned);
  void DFS_write(unsigned, vector<unsigned> &);
  void setDFS();
  void reset_visit() const;
  void split(const string &s, const char *delim, vector<string> &v);

rivate:
  ofstream *_simLog;
  int m, i, l, o, a;
  int gatenum;                    // maximum number of propable gates
  std::vector<unsigned> PI_list;  //gate IDs of PIs
  std::vector<unsigned> PO_list;  //gate IDs of POs
  std::vector<unsigned> DFS_list; //gate IDs through DFS traversal
  std::vector<vector<int>> FEC_pair;
  CirGate *gatelist;
  HashSet<CirGate> _cirHash;
  bool ever_sim;
```

m, i, l, o, a 為 aag head 右邊的數據，gatenum 是 m+o 的值，一旦輸入 CirRead 指令 gatelist 就會動態配置出 gatenum 個 CirGate 的陣列，並在 read 時先建立 DFSlist，而 gatelist 的 index 就是 ID，PO 及 PI 還有 DFS 存的就是 index，也就是 Gate 的 ID，以類似 hash 的方式 access 我欲取得的 Gate 減少時間複雜度。

而 FEC_pair 是紀錄每個 FEC pair，至於 ever_sim 則是判斷該 CirMgr 是否已經有模擬過(FEC_pair 會記住每一次模擬結果作調整)。

Algorithm :

1. CirSWeep :

```
void CirMgr::sweep()
{
  CirGate *temp = new CirGate[gatenum];
  for (int i = 0; i < DFS_list.size(); i++)
    temp[DFS_list[i]] = gatelist[DFS_list[i]];

  for (int i = 0; i < gatenum; i++)
  {
    if (gatelist[i]._type != temp[i]._type)
    {
      if (gatelist[i]._type == AIG_GATE)
      {
        cout << "Sweeping: " << gatelist[i].getTypeStr() << "(" << i << ") removed..." << endl;
        a--;
        erase_gate(gatelist[i]);
      }
      else if (gatelist[i]._type == UNDEF_GATE)
      {
        cout << "Sweeping: " << gatelist[i].getTypeStr() << "(" << i << ") removed..." << endl;
        gatelist[i] = CirGate();
      }
    }
  }

  delete[] temp;
  temp = 0;
}
```

先做一個只有 DFSlist 的值 temp 和真正的 gatelist 做比較，兩個 list 如果 type 不同則表示 gatelist 裡面的值是不在 DFS 裡面的，這時候就要將他除去(如果是 UNDEF 就只要將他清空即可)，因為可讀性以及更改方便，個人將除去的動作包成一個 function 叫做 erase_gate，其內容除了將該除去的 Gate 清空以外，也將他兩個 fanin 對應到的 Gate 的 fanout 名單將除去的 Gate 給 erase 掉。

如果 erase_gate 動作的時間算是一個 O(1)的話，整體時間複雜度應該是 O(n) (吧?)

2. CirOptimize :

主要可以歸類為兩種事件，其一是 fanin 相同或是剛好相反的 gate，另一則是處理 fanin 至少有一個是 const 的狀況，然而這兩個事件並非互斥，也就是說，同時有可能發生兩個 fanin 同時為 const 0(或 const 1)或是一個為 const 0 另一個則為 cons 1 的狀況，為避免互相干擾迴圈中兩個處理過程互相干擾，而且因為變化過程裡面有可能本來不是在兩種事件內的 gate 因為前面 gate 處理完之後而產生需要被 optimize 的狀況，所以我選擇在 DFSlist 中先處理相同 fanin 的情況。

```
void CirMgr::optimize()
{
  for (int i = 0; i < DFS_list.size(); i++)
  {
    int idx = DFS_list[i];
    /*----------------------------merge same fanins----------------------------*/
    if (gatelist[idx]._type == AIG_GATE)
    {
      if (gatelist[idx].fanins[0] / 2 == gatelist[idx].fanins[1] / 2)
      {
        if (gatelist[idx].fanins[0] == gatelist[idx].fanins[1])
        {
          cross_connect(gatelist[idx].fanins[0], idx);
        }
        else
        {
          cross_connect(0, idx);
        }
      }
    }
    /*----------------------------merge same fanins----------------------------*/

    /*----------------------------process const fanins----------------------------*/
    if (gatelist[idx]._type == AIG_GATE)
    {
      if (gatelist[idx].fanins[0] / 2 == 0)
      {
        if (gatelist[idx].fanins[0] % 2) //if invert
          cross_connect(gatelist[idx].fanins[1], idx);
        else
          cross_connect(0, idx);
      }
      else if (gatelist[idx].fanins[1] / 2 == 0)
      {
        if (gatelist[idx].fanins[1] % 2)
          cross_connect(gatelist[idx].fanins[0], idx);
        else
          cross_connect(0, idx);
      }
    }
    /*                              process const fanins                              */
```

因為將 gate 除去並將其輸入輸出重新對接的動作重複，便以 cross_connect 來實行，需注意的地方是根據 gate 的輸入是否 invert 以及其輸出是否 invert 在做對接的時候需要特別去考慮，自己在測試 ref 的過程中，發現他的 optimize 也是先 merge 相同 fanin 的 gate，但順序如果反過來不知道是否會影響其結構。

3. CirSTRash：

搜尋相同 fanin 的作法如果沒有使用一個讓搜尋方便的結構的話，搜尋時間的複雜度約略是 n 平方，而使用 hash 的話可以做到 n，所以在實行 strash 時，自己是暫時創造出一個簡單的 hash，結構上的設計，hash function 是將 gate 的兩個 fanin 的 literal 做相加，其相加的值就是 hash 結構中的 index，在 hash 的 head 檔案中，有 insert 以及一個 bool function 來將一個 row 中相同 fanin 的 gate 挑選出來。

```
size_t numBuckets() const { return _numBuckets; }

vector<Data> &operator[](size_t i) { return _buckets[i]; }
const vector<Data> &operator[](size_t i) const { return _buckets[i]; }

bool insert(const Data &d, Data &cmp)
{
  size_t idx = bucketNum(d);
  for (int i = 0; i < _buckets[idx].size(); i++)
  {
    cmp = _buckets[idx][i];
    if (check_in_same(d, cmp))
      return true;
  }
  _buckets[idx].push_back(d);
  return false;
}

bool check_in_same(Data d1, Data d2)
{
  if (d1.fanins[0] == d2.fanins[0] && d1.fanins[1] == d2.fanins[1])
    return true;
  else if (d1.fanins[0] == d2.fanins[1] && d1.fanins[1] == d2.fanins[0])
    return true;
  return false;
}

rivate:
// Do not add any extra data member
size_t _numBuckets;
vector<Data> *_buckets;

size_t bucketNum(const Data &d) const
{
  return (d.fanins[0] + d.fanins[1]);
}
```

```
void CirMgr::strash()
{
  _cirHash = HashSet<CirGate>(gatenum);
  vector<unsigned> _route_list;
  for (int i = 0; i < DFS_list.size(); i++)
  {
    int idx = DFS_list[i];
    if (gatelist[idx]._type == AIG_GATE)
    {
      CirGate cmp;
      if (_cirHash.insert(gatelist[idx], cmp))
      {
        int idx_mer = cmp._ID;
        cout << "Strashing: " << idx_mer << " merging " << idx << "..." << endl;
        merge_gate(gatelist[idx_mer], gatelist[idx]);
      }
    }
  }
  _cirHash.reset();
  setDFS();
}
```

當 insert 進去 hash 的 gate 發現 hash 中有相同 fanin 的 gate 時會回傳 true，這時候 cmp 的 gate 就會是最早存進 hash 並且擁有相同輸入的 gate，後則將被判定同樣輸入的 gate 給 merge 掉，merge 的 function 除了將被 merge 的 gate 的輸出全部推進另一 gate 外，還要將輸出的 fanin 接回剩下的 gate，需要在另外判斷是否有 invert 存在。

4. CirSImulate :

以 filesim 為例(和 random 大同小異)，自己在設計每個 gate 的 sim value 上並不是使用 size_t 而是直接使用 bitesets 來做邏輯運算(個人覺得比較直觀)，另外 filesim 主要可以分成幾個階段，讀取檔案做 error handle 後將 dfs 所有的 gate 做模擬，最後創造一個 hash 結構(設計上寫成另一個 class 並放在 hashmap 的 head 檔案裏面)尋找具有共同 simvalue 的 gate(其結構與 strash 類似)，並重新更新 FEC pair，其時間複雜度約略是 n(雖然差 ref 很多)

```
void CirMgr::fileSim(ifstream &patternFile)
{
  /*------------------------------read stream-----------------------------*/
  vector<bitset<64>> bits_data;
  vector<string> bits_str;
  bits_str.resize(i);
  bits_data.resize(i);
  string inputStr;
  int count = 0;
  int sim_num = 0;
  bool do_error = false;
  while (getline(patternFile, inputStr))
  {
    vector<string> token;
    split(inputStr, " ", token);
    inputStr = token[0];
    if (inputStr.size() != i)
    {
      cerr << "\nError: Pattern(" << inputStr << ") length(" << inputStr.size() << ") does not match the number of inputs(" << i << ") in a circuit !!" << endl;
      do_error = true;
      break;
    }
    if (token.size() != 1)
    {
      inputStr = token[1];
      cerr << "\nError: Pattern(" << inputStr << ") length(" << inputStr.size() << ") does not match the number of inputs(" << i << ") in a circuit !!" << endl;
      do_error = true;
      break;
    }
    for (int i = 0; i < inputStr.size(); i++)
    {
      if (inputStr[i] != '0' && inputStr[i] != '1')
      {
        cerr << "\nError: Pattern(" << inputStr << ") contains a non-0/1 character('" << inputStr[i] << "')." << endl;
        do_error = true;
        break;
      }
      bits_str[i] = inputStr[i] + bits_str[i];
    }
    if (do_error)
      break;
    count++;
    sim_num++;
```

Error handle 中須特別留意的是讀進來的字串是否是有非 0/1 以外的字串，或是輸入的列數與 PI gate 的數量不符。

```
    }
    if (do_error)
      break;
    count++;
    sim_num++;
    if (count >= 64)
    {
      int bits_long = 64;
      /*------------------------------simulate PI------------------------------*/
      for (int i = 0; i < PI_list.size(); i++)
      {
        int idx = PI_list[i];
        bitset<64> temp(bits_str[i]);
        gatelist[idx].sim_bits = temp;
      }
      /*------------------------------simulate PI------------------------------*/

      /*------------------------------operate simulation------------------------------*/
      operate_sim();
      /*------------------------------operate simulation------------------------------*/

      /*------------------------------output simlogfile------------------------------*/
      if (_simLog)
        write_simlog(bits_long);
      /*------------------------------output simlogfile------------------------------*/
      count = 0;
      for (int i = 0; i < PI_list.size(); i++)
      {
        bits_str[i].clear();
      }
    }
  }
```

設計上每讀入 64 個 bits 數量就會 sim 一次，其中 operate_sim()做的事情有模擬各個 gate 以及輸入或是更新 sim 的 FEC pair，其中，被 sim 的 gate 不用擔心自己的 fanin 還沒被 sim 過(因為是 DFS 路徑)，另外創造 FEC pair 的 hash，決定 index 的機制是取 gate 的 simvalue 值在依據 bucket 大小取餘數，如果 simvalue 的互補值比較小則使用互補值，而至於更新 FEC pair 則可以視為是尋找每個 pair 中是否還有更小的 FEC pair 的子問題，然後分出來的子 FEC pair 需要 sort。

5. CirFraig：

嗯...來不及寫了，不過可以大概描述一下如果要寫會怎麼構思，主要是將每個 DFS 的 gate 做一次 sim 之後得出其 FEC pair，然後將 FEC pair 做 SAT 論證，在依據論證出來的兩種結果做出不同的對應，在更新 FEC pair。