# Final Project Report
# of *DATA STRUCTURES & PROGRAMMING*

Jian-Han Wu 吳建翰 B06901086

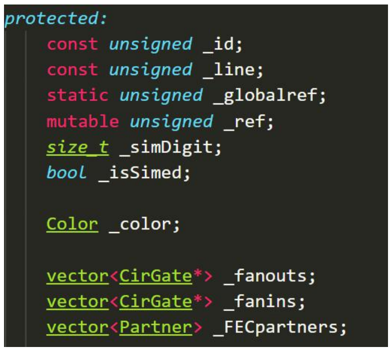
EE3 b06901086@ntu.edu.tw

## I. DATA STRUCTURES

There are two classes describing gates, *CirMgr* and *CirGate*. The former is to manage the basic information of whole circuit, including number of input gates, output gates, AIG gates and total number of gates (i.e. five numbers in the first line of an aag file) and is responsible for any optimization process, including sweeping, basic optimization, structural hashing (*strash*), simulation and fraig. In order to implement these functions, there is a vector array, **vector <CirGate*>** *_vgate*, that stores all of the gates in the circuit according their IDs. For example, *_vgate [3]* points to the gate with ID 3. Also, there are several arrays that store the information of the circuit, consisting of an array that stores IDs of gates with any *NIL* fanin, an array that stores IDs of gates defined but not used, a DFS list and FEC pair list.

On the other hand, CirGate saves basic information about a particular gate, as referred to Figure 1, including its ID, its line number in aag file, its color (for DFS), its simulation pattern, its FEC partners, the fanouts and fanins of the gate etc. Most of these data members are defined in private or protected, meaning that it is not permitted to be used in outer space. Therefore, there are numerous public functions that return values of these data members or edit values of them. In order to distinguish different gates like inputs, outputs, constant gates and AIG gates. Inheritance is applied to the class and polymorphism as well.

The vector array *_vgate* acts as an important role for *CirMgr* to access gates defined in *CirGate*. After an aag file is read in, all gates are pushed back to the array. Note that not all of the IDs are used. Thus the array may contain some index that has a null pointer and it is necessary to be handled in my program. The element of the array is of the type of pointer *CirGate*. The advantage of using pointers is that there is no extra memory used for copy. The same concept is also applied to the DFS list. In nil fanin list and not used list, however, store IDs of the gate instead due to the convenience for the command *cirp -fl*.



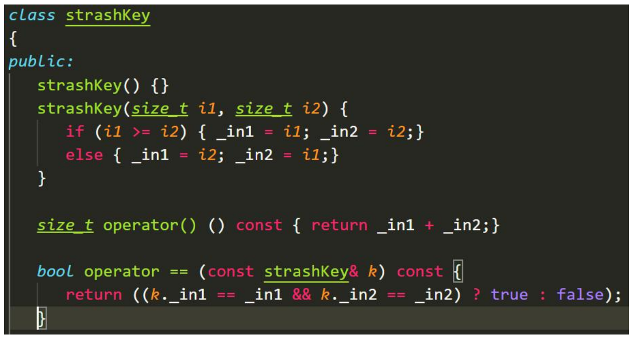
```
protected:
    const unsigned _id;
    const unsigned _line;
    static unsigned _globalref;
    mutable unsigned _ref;
    size_t _simDigit;
    bool _isSimed;

    Color _color;

    vector<CirGate*> _fanouts;
    vector<CirGate*> _fanins;
    vector<Partner> _FECpartners;
```

Figure 1. Data members in CirGate.

To implement the function *strash*, a hash map class *HashMap* is designed to hash all AIG gates in DFS list by their fanins. *HashMap* is in the template of *<Hash Key, Hash Data>*. The following is a hash key class in "*myHashMap.h*", *strashKey*. Two fanins of a gate are parameters to the constructor and the strash key is generated by summing them up. To detect whether two gates are the same on their structures (namely, with same fanins), the operator "==" is overloaded if two fanins are the same. Note that when constructed, the object sorts fanins such that _in1 > _in2 for all strash keys.



```
class strashKey
{
public:
   strashKey() {}
   strashKey(size_t i1, size_t i2) {
      if (i1 >= i2) { _in1 = i1; _in2 = i2;}
      else { _in1 = i2; _in2 = i1;}
   }

   size_t operator() () const { return _in1 + _in2;}

   bool operator == (const strashKey& k) const {
      return ((k._in1 == _in1 && k._in2 == _in2) ? true : false);
   }
```

Figure 2. Implementation of strashKey

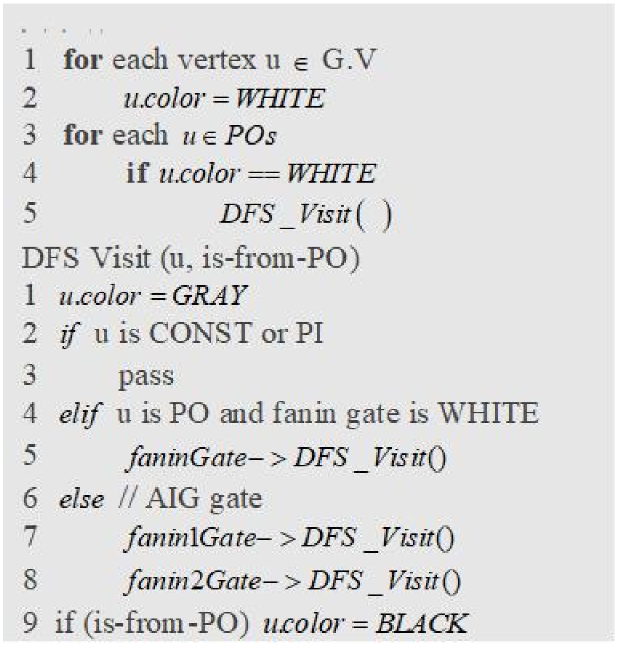
Functions in HashMap includes check, insert, update, query, and delete. Hash Data is CirGate*. Thus the hashing is actually hashing gates into the map according to their inputs.

## II. ALGORITHMS & ANALYSIS

The following shows algorithms of some optimization functions.

### *A. Sweep*

The sweep function deletes not used gates in the circuit. Not used gates are those which do not reach POs (It's a little different from the definition in the project document). Therefore, what the function does is to delete all AIGs that are not in the DFS list.



```
1  for each vertex u ∈ G.V
2        u.color = WHITE
3  for each u ∈ POs
4        if u.color == WHITE
5              DFS_Visit( )
DFS Visit (u, is-from-PO)
1  u.color = GRAY
2  if u is CONST or PI
3        pass
4  elif u is PO and fanin gate is WHITE
5        faninGate->DFS_Visit()
6  else // AIG gate
7        fanin1Gate->DFS_Visit()
8        fanin2Gate->DFS_Visit()
9  if (is-from-PO) u.color = BLACK
```

Color is assigned while DFS is applied. DFS is defined in the function "*CirMgr::DFSTrace*". The algorithm of depth first search is defined above. Note that in DFS Visit, a gate is colored *BLACK* if it can be traced by any POs. Thus, it is to be sure that all of the gates in the DFS list is colored *BLACK* while the others are colored *GRAY*. Therefore, how to delete gates that are not used is exactly to delete those with color *GRAY*.

After removing a gate, it is necessary to alter the fanouts list for gates in DFS list. Also, the not-used-gate list is to be altered to PIs that are not used (also colored *GRAY*). What I did in my program is to check fanins of the AIGs that are to be deleted. However, a faster way to update fanouts list of gates in DFS list is to check the fanouts of gates in the DFS list. If any gate in the fanouts list is *GRAY*, just delete it. Last but not least, the memory occupied by the deleted gate should be released, and hence the operator *delete* is used and *_vgate [the gate]* = 0.

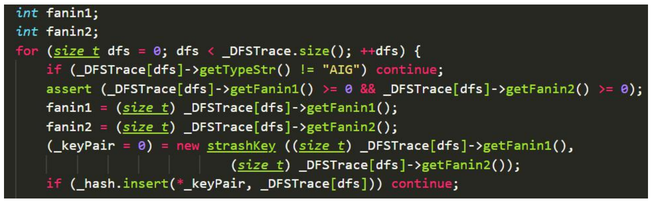
The sweep function works very fast since once all of the gates are checked, the work will be done. Time complexity is *O(n)*.

### *B. Simple Optimization*

There are four cases to be handled in the function *optimize ()* as being referred in the document. The mutual task is to decide which fanin of the deleted gate "inherits" fanouts of the deleted gate and vice versa. After that, the DFS list is to be updated. The algorithm is to redo DFS and we can get a new list.

### *C. Structural Hash*

The implementation of strash is mentioned before. The following figure shows the code in function *strash* and it is the part that hashes gates into the hash map. The bottle neck of this function is to update fanouts list of every gate deleted, which forms the second nest for loop. However, it works still fast.



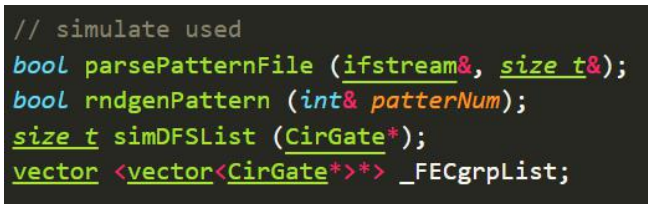
```
int fanin1;
int fanin2;
for (size_t dfs = 0; dfs < _DFSTrace.size(); ++dfs) {
    if (_DFSTrace[dfs]->getTypeStr() != "AIG") continue;
    assert (_DFSTrace[dfs]->getFanin1() >= 0 && _DFSTrace[dfs]->getFanin2() >= 0);
    fanin1 = (size_t) _DFSTrace[dfs]->getFanin1();
    fanin2 = (size_t) _DFSTrace[dfs]->getFanin2();
    (_keyPair = 0) = new strashKey ((size_t) _DFSTrace[dfs]->getFanin1(),
                                     (size_t) _DFSTrace[dfs]->getFanin2());
    if (_hash.insert(*_keyPair, _DFSTrace[dfs])) continue;
```

Figure 3. strash implementation

How to remove an element in a vector? We encounter so many times when we update vectors such as nil fanin gates, not used gates etc. My algorithm is to use the last element to cover the one to be deleted, and then pop the vector. After popping back, the counter of the for loop should minus one since the last element might be the one to be deleted as well.

*D. Simulation*

There are several additional data members and functions defined in CirMgr.h, as shown in the figure below.



```
// simulate used
bool parsePatternFile (ifstream&, size_t&);
bool rndgenPattern (int& patterNum);
size_t simDFSList (CirGate*);
vector <vector<CirGate*>*> _FECgrpList;
```

Figure 4. Additional data members.

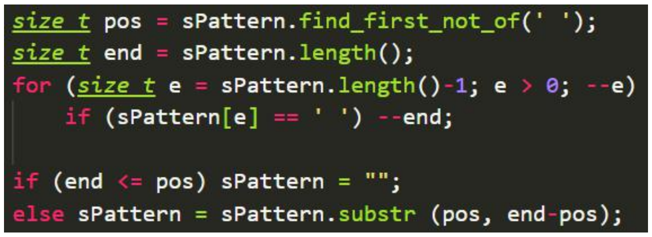
The two Boolean functions, "parse-Pattern-file ()" and "rnd-gen-Pattern ()" set 64-bit input patterns when they are called by file-Sim and random-Sim, respectively. The following paragraphs show details of the simulation procedure.

*1) Initializing & Resetting* Since the program simulates 64 bits every time. When it encounters a new cycle, all of the gates that are simulated before need to refresh its simulation digits and a Boolean variable to check whether a gate is simulated, which are defined as $\_simDigit$ and $\_isSimed$ in "CirGate.h".

*2) Simulation Pattern* The simulation pattern must be a 64-bit one, which is an 8-byte pattern. According to the data type of variables, an unsigned long-long integer is 8 bytes. Therefore, we use a size_t variable to describe a simulation pattern. As for the operation of the pattern, it needs no worry thanks to bit-wise property of binary operators "&", "|" and "^", meaning "*AND*", "*OR*" and "*XOR*" respectively. The negation of a pattern is a pattern that changes 0 to 1 and 1 to 0 in the original pattern, which is exactly the XOR operation with the maximum number of *size_t*, **ULLONG_MAX**.

*3) Reading Pattern File* The method to read a pattern file is defined in $parsePatternFile\left(ifstream\&,size\_t\&\right)$. The first parameter is the input file, and the second one is a line counter that counts the number of lines. Once the counter reaches 64, it breaks the while loop that reads lines and return true. This variable also determines whether the file reaches the end. The variable is set to 0 every time the function is called. If the file ends, the parameter in the while loop $getline\left(patternFile,line\right)$ returns false and thus terminates the loop. At this moment, the line counter equals to zero, and the whole function returns false.

While getting a line, the line needs to be altered first since it may contain some spaces in the front or in the end of the line. The following figure shows the process of obtaining the sub string of the original one.

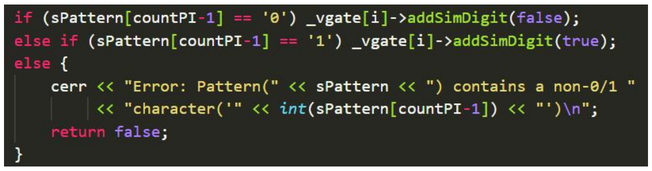


```
size_t pos = sPattern.find_first_not_of(' ');
size_t end = sPattern.length();
for (size_t e = sPattern.length()-1; e > 0; --e)
    if (sPattern[e] == ' ') --end;

if (end <= pos) sPattern = "";
else sPattern = sPattern.substr (pos, end-pos);
```

Figure 5. Code that trims the pattern string.

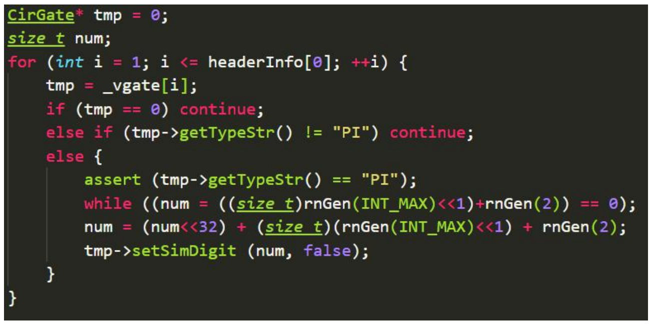
*4) Set input patterns* Since there might be abundant of PIs in the circuit and it is not practical that we created the pattern numbers in *size_t* if the number of PIs is large. As a result, a binary adder is defined in CirGate. For every PI gate, encountering a 0 pattern, becomes twice of its original value while encountering 1, the simulation pattern should twice and plus 1. Figure 6 shows how the adder works. The parser is added at the end of the block to detect if there is any non-binary digit.



```
if (sPattern[countPI-1] == '0') _vgate[i]->addSimDigit(false);
else if (sPattern[countPI-1] == '1') _vgate[i]->addSimDigit(true);
else {
    cerr << "Error: Pattern(" << sPattern << ") contains a non-0/1 "
         << "character('" << int(sPattern[countPI-1]) << "')\n";
    return false;
}
```

Figure 6. Adder.

*5) Random Pattern Generator* The random case is quite different from the read-file case and is much easier, however, because we can define a random size_t pattern for any PI gates. The only limitation is that the random Generator can only generate an *integer* rather than *size_t*. Note that the integer is a four-byte number with its MSB 0 to indicate positive. In order to generate a *size_t* pattern, the $rnGen(INT\_MAX)$ is applied twice to generate two signed integers and $rnGen(2)$ to make up the MSB problem (Note: the implementation shifts the 31-bit pattern by one and add the remaining digit). To set the pattern to the PIs, a function *setSimDigit* is applied, containing the pattern to be set and a Boolean to determine whether to reset the gate. Once the reset is *true*, the gate is to be reset and the pattern is equal to zero.



```
CirGate* tmp = 0;
size_t num;
for (int i = 1; i <= headerInfo[0]; ++i) {
    tmp = _vgate[i];
    if (tmp == 0) continue;
    else if (tmp->getTypeStr() != "PI") continue;
    else {
        assert (tmp->getTypeStr() == "PI");
        while ((num = ((size_t)rnGen(INT_MAX)<<1)+rnGen(2)) == 0);
        num = (num<<32) + (size_t)(rnGen(INT_MAX)<<1) + rnGen(2);
        tmp->setSimDigit (num, false);
    }
}
```

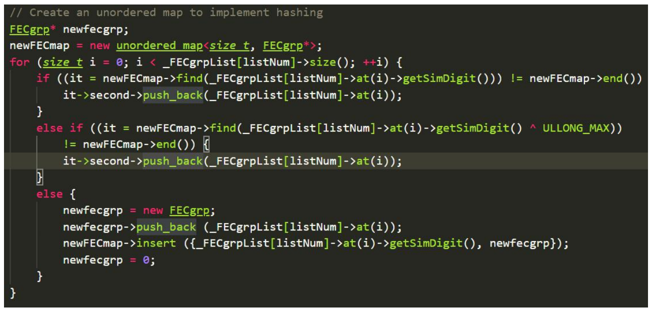
Figure 7. Random pattern

*6) Patterns for AIGs* To set patterns of AIG gates, *simDFSList*() simulates those gates using the DFS list. The function is called recursively. If a gate is simulated, return its pattern; otherwise, find its fanin gate and call the function to find the pattern of the fanin gate. If the gate contains two valid (i.e. defined) gates, just call the function on these gates and do AND operation after receiving patterns. If a fanin is reversed, XOR the pattern with $ULLINT\_MAX$. The Boolean $\_isSimed$ acts as a significant role here. Without to determine whether a gate is simulated, the DFS will go through the whole circuit repeatedly until it reaches PIs.

*7) FEC Hashing* Now, after simulating all gates in the DFS list, these gates contain a simulate pattern. If some of the gates have the same simulate patterns, they are in the same FEC group. To find all FEC groups, hashing, again, is to be applied.

I use the *STL container* unordered map here in replace of the hash map. First, I push all the gates into an FEC group, and push the FEC group into a FEC group list defined in *CirMgr*.

In the while loop, let's suppose there are several FEC groups already in the FEC group list, but some of the groups might contain gates that are not all the same but under a given pattern they generate the same simulation patterns in the previous simulation. After the simulation this time, the patterns might alter and thus be different from others. Therefore, the first thing is to detect whether gates in the FEC group needs to be classified. If yes, the classification would be done.



```
// Create an unordered map to implement hashing
FECgrp* newfecgrp;
newFECmap = new unordered_map<size_t, FECgrp*>;
for (size_t i = 0; i < _FECgrpList[listNum]->size(); ++i) {
    if ((it = newFECmap->find(_FECgrpList[listNum]->at(i)->getSimDigit())) != newFECmap->end())
        it->second->push_back(_FECgrpList[listNum]->at(i));
    }
    else if ((it = newFECmap->find(_FECgrpList[listNum]->at(i)->getSimDigit() ^ ULLONG_MAX))
        != newFECmap->end()) {
        it->second->push_back(_FECgrpList[listNum]->at(i));
    }
    else {
        newfecgrp = new FECgrp;
        newfecgrp->push_back (_FECgrpList[listNum]->at(i));
        newFECmap->insert ({_FECgrpList[listNum]->at(i)->getSimDigit(), newfecgrp});
        newfecgrp = 0;
    }
}
```

Figure 8. FEC group classification.

The above picture shows the process of hashing these gates. First, we create an unordered map and hash gates in this FEC group according to their patterns. Note that a pattern A is seen to be the same as a pattern B if and only if they have the same simulation patterns or if and only if they have patterns that are actually in reversed, that is,

$$patternA = partternB \wedge ULLINT\_MAX .$$

If a pattern is found to be the same with patterns in a particular bucket in the hash map, insert the gate into the bucket. If no buckets match, create a new bucket and insert the gate into it. Note that the insert process needs a parameter $FECgrp*$, which is defined in replace of $vector < CirGate* >$. Thus, if a new FEC group is created, the gate should be pushed into the vector and the vector then be inserted into the map. After the classification, the FEC group list needs update. First, the original FEC group that is classified just now should be deleted first, and those new FEC groups in the unordered map should be inserted into the FEC group list.

After the simulation patterns are all done. The FEC partners are assigned to every gate.

The performance is too slow, however. It needs too much time for sim13.aag. The bottleneck might be the pattern setting since there are too many gates here.

## III. EXPECTATIONS

I cannot complete the last operation fraig due to the limit of time. What I want to do is to learn from my classmates code in order to know what I still have to learn and try my best to make my program work faster and more stable.

- END OF THE DOCUMENT -