2018-1
Data Structure and Programming
Final Project Report

Student: 電機二 B05901003 徐敏倩
Email: b05901003@ntu.edu.tw

Table of Contents

I. Type definition

A. typedef vector<CirGate*> GateList
B. typedef vector<int> IdList
C. typedef vector<size_t> PointerInvList
D. typedef unordered_map<size_t, CirGate*> StrashHash
E. typedef unordered_map<size_t, PointerInvList*> SimHash
F. typedef map<CirGate*, Var> fraigMap

For all parts that we are supposed to use HashMap, instead of writing a new class on my own, I choose to use unordered_map in the C++ Standard Library.

II. Data members of classes

A. class CirGate

1. GateType _type
   Store the type of the CirGate.
   It may be UNDEF_GATE, PI_GATE, PO_GATE, AIG_GATE or CONST_GATE.

2. CirGate* _in[2]
   Store the pointer of its input CirGate.
   For different type of CirGate, the number of its input CirGate differs. If it is UNDEF_GATE, PI_GATE or CONST_GATE, since it doesn't have any input, that is the number of its input gate is zero. Therefore, _in[2] doesn't store anything. If it is PO_GATE, it has only one input. That is, _in[1] points to its input, while _in[0] stores nothing. If it is AIG_GATE, it has two inputs. Hence, _in[0], _in[1] points to the two inputs respectively.

3. GateList _out
   Store the pointer of the output CirGate.
   The outputs of the CirGate can be any number. That is, _out can be in any size. However, if the type of the CirGate is PO_GATE, _out should be empty since it is the output gate and shouldn't have any output gates.

4. char _inSign[2]

_inSign	Inverted	UNDEF_GATE	Mark
'0'	1	0	!
'1'	1	1	*!
'2'	0	0	
'3'	0	1	*

   Store the linking information of the current CirGate and its inputs.
   Elements in the _inSign can only be '0', '1', '2', or '3', which represents different linking information. The linking information is listed as below.

5. int _gateId
   Store the gate id of the CirGate.

6. int _lineNo
   Store which line in the input aag file is the CirGate in.
   It is trivial for PI_GATE, PO_GATE, and AIG_GATE since they are all stated in the input aag file. However, for CONST_GATE and UNDEF_GATE, automatically set _lineNo to zero, since they won't be stated in any line of the input aag file.

7. string _name

Store the name of the CirGate.
Name of the PI_GATE and PO_GATE may be specified in the input aag file, while there won't be any name definition for AIG_GATE, CONST_GATE or UNDEF_GATE.

8. mutable size_t _mark
   Store the information whether or not the CirGate has been traverse.
   When _mark equals to GlobalRef, meaning that the CirGate has already been traverse this time and certain action should take place coordinately.

9. bool _dfs
   Store the information whether the CirGate is in the dfsList.

10. size_t _simVal
    Store the information of the simulating value.

11. IdList _fecGate
    Store the _gateId of the CirGate that is currently in the same FECGroup as the CirGate.

B. class CirMgr

1. GateList _totalList
   Store all the CirGate in the circuit.
   CirGates in the _totalList are in the certain order, PI_GATE, PO_GATE, AIG_GATE, CONST_GATE, UNDEF_GATE. Therefore, without saving other list like _PIList, _POList, and so on, I can still easily traverse through different type of CirGates.

2. map<int, CirGate*> _sortList
   Store all the CirGate in the order of its _gateId.
   Since _totalList isn't sorted in the order of id, it's hard to find a CirGate by its _gateId. Therefore, save _sortList so that we can get the pointer of the CirGate by its _gateId faster.

3. mutable GateList _dfsList
   Store the CirGate in dfs order.
   It is empty in the beginning and it will only be constructed after dfs() evokes.

4. mutable GateList _AIGdfsList
   Store the CirGate which is AIG_GATE in dfs order.
   It is empty in the beginning and it will only be constructed after dfs() evokes.

5. int _m, _i, _l, _o, _a
   Store the header of the aag file.
   _m, _i, _l, _o won't be alter throughout the whole process. However, the value of _a might be changed when any of the AIG_GATE being deleted.

6. mutable size_t _dfsMark
   Store the information whether the _dfs of each CirGate has been updated.
   When _dfsMark equals to GlobalMark, meaning that the the _dfs of each CirGate has been updated to the latest situation.

7. vector<PointerInvList*> _fec
   Store all the FECGroup.

8. ofstream *_simLog
   The pointer of the output file when writing the _simVal.

III. Important Self-define Member functions of class CirGate

A. Simulate

1. void simPO()
   Simulate the PO_GATE from its input.
   Simulation can be divided into two part by whether or not the input of it is inverted. If so, invert the input signal and assign it to the PO_GATE.

2. void simAIG()
   Simulate the AIG_GATE from its inputs.
   Check whether the inputs are inverted and correspondingly 'AND' the two signals (bitwise calculation).

3. void setPIsim(char c)
   Set the _simVal of the PI_GATE.
   To add a new signal c, multiple the original signal by two then add the corresponding signal.

4. void setPIrandsim(size_t i)
   Set 64 signals the _simVal of the PI_GATE in once, i.e. _simVal equals to i.

5. void updateSim(size_t i)
   Update the _simVal by multiply the original _simVal by 2 and add the new one, which is (size_t) i.

6. void resetSimVal()
   Reset the _simVal to zero.

7. void setFECGate(PointerInvList u)
   Set _fecGate, id of the CirGate which is in the same FECGroup as the current CirGate, to be the CirGate in PointerInvList u expect itself.
   Supplementary, the id of the CirGate might be negative if the current CirGate is in the same group with its inverse.

8. void resetFECGate()
   Empty the _fecGate of the CirGate.

B. Reset inputs and outputs

1. void removeOut(CirGate* s)
   Remove CirGate* s from the output list _out.

2. void removeInofOut(CirGate* prev, bool inverse)
   Change the input of all the outputs that point to itself to CirGate* prev and set its _inSign corresponding to bool inverse.

3. void changeINofOUT(CirGate* replace, bool bubble = 0)
   Change the input of all the outputs that point to itself to CirGate* replace and inverse the corresponding _inSign if bool bubble is true.

4. void changeIN(int index, CirGate* replace, bool bubble)
   Change the one of the input, determined by int index, to replace and inverse the corresponding _inSign if bool bubble is true.

5. void inSigninverse(int index, bool bubble)
   Inverse the corresponding _inSign.

C. Optimization

1. bool OptCase(CirGate*& prev, bool& zero, bool& inverse)
   Determine the optimization case of CirGate.
   Return false when it doesn't fall into any categories. Otherwise, do the following.
   a) If one of the fanins of an AND gate is a constant 1
      Set CirGate*& prev to the other input of the AND gate. Remove the current CirGate from the output of 'prev', add all the outputs of the current CirGate to the output of 'prev', and reset the input of the outputs of current CirGate to 'prev' correspondingly.

   b) If one of the fanins of an AND gate is a constatnt 0
      Set bool& zero to 1, bool& inverse to 0, and CirGate*& prev to the other input. Then, remove the current CirGate from the output of 'prev'.

   c) If both fanins of an AND gate are the same
      Set CirGate*& prev to be the input of the AND gate. Remove the current CirGate from the output of 'prev', add all the outputs of the current CirGate to the output of 'prev', and reset the input of the outputs of current CirGate to 'prev' correspondingly.

   d) If one of the fanins of an AND gate is inverse to the other fanin
      Set bool& zero to 1, bool& inverse to 0, and CirGate*& prev to be the input. Then, remove the current CirGate from the output of 'prev'.

IV. Sweep

Sweep out all the AIG_GATEs and UNDEF_GATEs that is not in the _dfsList. That is, these gates can't be reached by PO_GATEs.

Realize it by using a function 'void CirMgr::sweep()' and its pseudo code is as follows.

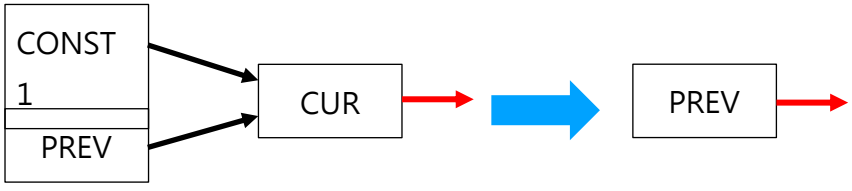
```
void CirMgr::sweep()
   if (DFS information is not updated)
      for each_gate in total_List(gate)
         gate->reset_dfs_information;

      dfs()

      DFS information is updated

   for each_gate in total_List(gate)
      if (gate is AIG_GATE or UNDEF_GATE)
         if (gate is AIG_GATE)
            if (gate->fanin0 != 0) remove gate from gate->fanin0 output list
            if (gate->fanin1 != 0) remove gate from gate->fanin1 output list
         delete gate
         remove gate from all lists
```

V. OPTimize

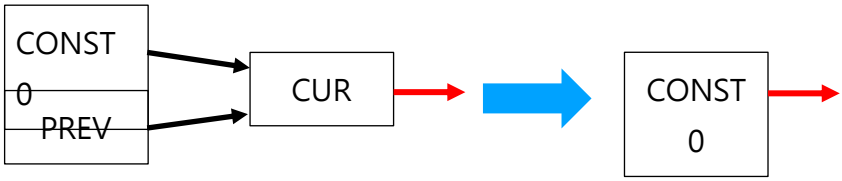
Perform trivial circuit optimizations including constant propagation and redundant gate removal.

There are following four cases that will be detected in this.

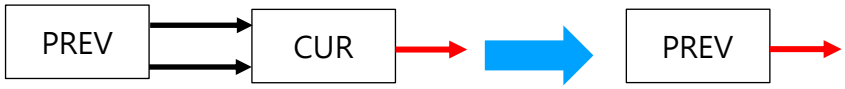
A. If one of the fanins of an AND gate is a constant 1



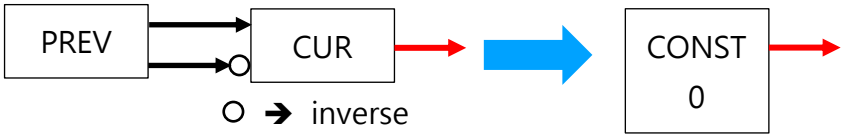
B. If one of the fanins of an AND gate is a constatnt 0



C. If both fanins of an AND gate are the same



D. If one of the fanins of an AND gate is inverse to the other fanin



Realize it by using a function 'void CirMgr::optimize()'. However, the main part of the is in the CirGate::OptCase(CirGate*&, bool&, bool&).
In order to get the job done, for every AIG_GATEs in the _dfsList, call CirGate::OptCase(CirGate*&, bool&, bool&), capability of the function been mentioned in the content above which it decide the case and update the linking of the CirGate if the current CirGate is not replaced by 'CONST 0'. Finish the case that 'CONST 0' replaces the gate by setting the input of the original outputs to be 'CONST 0'. Finally, erase the CirGate that should be remove from the _totalList and the _sortList.

VI. Strash

Merge the structurally equivalent gates, which is having the same input gates and same signs.

In order to detect the equivalence faster, I use StrashHash, which is unordered_map<size_t, CirGate*>, to store the information of the gates. For each AIG_GATEs traversing in the dfs order, its key is set up as shown below.

```
in0 = 2 * gate->id_of_fanin0 + gete->inverse_of_fanin0
in1 = 2 * gate->id_of_fanin1 + gete->inverse_of_fanin1
key = (smaller of the (in0, in1)) * 2_to_the_power_of_32 + the_other_one
```

Therefore, the key only equivalent with one another when the CirGate are structurally equivalent.

If there exists structurally equivalent gates, replace one of them by the other one and repeat the steps in the optimization for removing the gate.

VII. Simulation

By initializing the _simVal of the PI_GATEs and CONST_GATE, simulate the corresponding _simVal for AIG_GATEs and PO_GATEs, and separate the AIG_GATEs into different FECGroups. There are two simulating approaches. One of them initialize the gate by a pattern file which it calls 'void CirMgr::fileSim(ifstream& patternFile)' to realize, while the other initialize the gate randomly by calling 'void CirMgr::randomSim()'.

For ‘void CirMgr::fileSim(ifstream& patternFile)’, check for the correctness of the input pattern file and assign the _simVal of PI_GATEs correspondingly. Then, for every 64 input signals call ‘void CirMgr::sim()’ to simulate.

For ‘void CirMgr::randomSim()’, set the threshold to be 10, no matter what the number of the PI_GATEs is. That is, whenever the size of the fecGroups remains unchanged for ten times, simulation ended. In this function, ‘void CirMgr::sim()’ is also called for simulation.

In ‘void CirMgr::sim()’, first of all, simulate all the gates. Secondly, if the fecGroups is not empty, separate the gates in the same group by using SimHash and its _simVal as key. All gates originally in the same fecGroups will still be in the same fecGroups if they have the same _simVal or being totally different. For the case that fecGroups is empty, consider all of the gates is in the same fecGroups. Last, assign the result in the SimHash back to _fec, remove all the fecGroup with size 1, and sort the fecGroups and fecGroup in increasing order.

VIII. Fraig

Based on the fecGroups, perform fraig operations.

First of all, built up a circuit of all the gates that is in the _dfsList.

Secondly, go through the entire fecGroups and prove every pair of gates in the same fecGroup by SAT.

If the pair of gates has been proven to be the same, replace the one with larger _gateId by the other one.

Otherwise, record the input signals that can separate them. Then, keep on proving the equivalence of all gates.

Also, whenever it has recorded 64 signals that are able to separate more of the gates, simulate all the gate and reconstruct the fecGroups again.

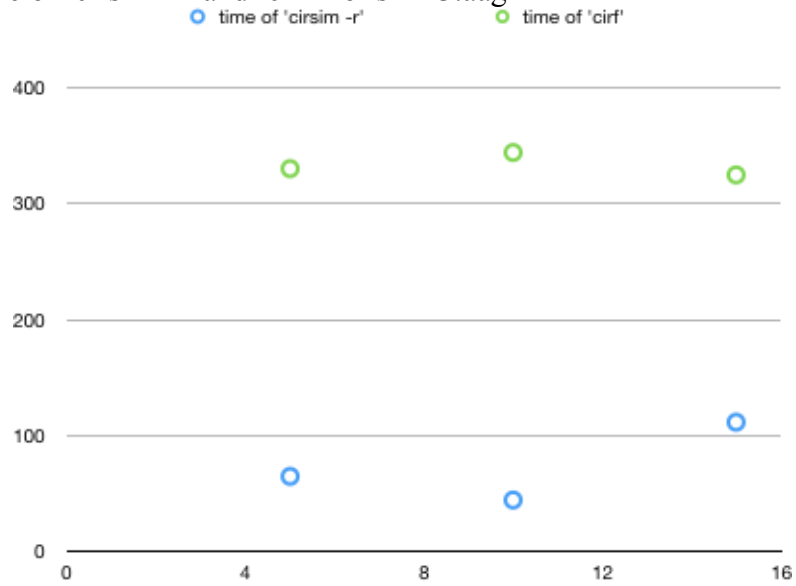
Repeat the previous steps, until the fecGroups to be empty.

Instead of choosing only to go through part of the fecGroups, I choose to go through all of them.

IX. Results and analysis of Experiments

A. Changing different threshold for ‘void CirMgr::randomSim()’

Plotting the time of ‘cirsim -r’ and ‘cirf’ for sim13.aag



Therefore, from the previous result, I choose to let threshold equal to 10, since it has the smallest time of ‘cirsim -r’.

X. Feedback

Before taking this course, I don' t really have experience in coding in c++. The first homework of the course was basically the first c++ code I've ever written. Although I was quite worried about whether I would be able to survived, I chose to challenge myself to take the course. Throughout the whole semester, I've spent a lot of time on the homework and did learn a lot of things. I really learn a lot from this course and think that this is one of the best course I have ever taken. Though every time I can't fixed the error, it was very frustrating. However, the sense of satisfaction when finishing the homework can really cover all the negative feelings.