資料結構與程式設計

Functionally Reduced And-Inverter Graph (FRAIG)

(Due: 9:00pm, Tuesday, Jan 14, 2014)

0. Objectives

1. Putting everything you learn in a project.
2. Using hash to detect structurally equivalent signals in a circuit.
3. Performing Boolean logic simulations and designing your own data structure to identify functionally equivalent candidate pairs in the circuit.
4. Being able to call Boolean Satisfiability (SAT) solver to prove the functional equivalence.
5. Writing a professional project report.

1. Problem Description

In this final project, we are going to implement a special circuit representation, FRAIG (Functionally Reduced And-Inverter Graph), from a circuit description file. The generated executable has the following usage:

fraig [**-File** <*dofile*>]

where the **bold words** indicate the command name or required entries, square brackets "[ ]" indicate optional arguments, and angle brackets "< >" indicate required arguments. Do not type the square or angle brackets.

This fraig program should provide the following functionalities:

1. Parsing the circuit description file in the AIGER format. This has already been covered in Homework #6. Please note that at this stage, DO NOT perform any optimization as described latter in this document.
2. Sweeping out the gates that cannot be reached from POs (See "`CIRSWeep`" command). After this operation, all the gates that are originally "defined-but-not-used" will be deleted. However, do not remove PIs even if any of them

becomes unused. (Note: these PIs will be added to the list of unused gates and be reported by the "cirg -fl" command)

3. Performing trivial circuit optimizations including: (a) If one of the fanins of an AND gate is a constant 1, this AND gate can be removed and replaced by the other fanin, (b) If one of the fanins of an AND gate is a constatnt 0, this AND gate can be removed and replaced by the constant 0 (Note: If the list of fanouts of the other fanin becomes empty, and if this other fanin is an AIG or PI, it will be added to the list of unused gates), (c) If both fanins of an AND gate are the same, this AND gate can be removed and replaced by its fanin, (d) If one of the fanins of an AND gate is inverse to the other fanin, this AND gate can be removed and replaced by a constant 0.

   Please note that different orders of optimization operations may result in different circuit structures. However, their functionalities should always remain the same. That is, the optimization procedure should not alter the functionality of the circuit or affect the correctness of the other operations.

   There should be a member function in `class CirMgr` to perform the above optimization. There should also be a command `CIROPTimize` to perform it. However, you should perform the above optimizations on the gates in the DFS list only. For the gates that are not in the DFS list (i.e. those which cannot be reached from POs), please skip them. However, if any of the gates got removed during these operations, you should update the DFS list in the end.

4. You should provide a function to perform "structural hash" (*strash*) on the circuit netlist. The strash operation is to merge the structurally equivalent signals (i.e. replace a gate with its functionally equivalent one) by comparing their gate types and permuting their inputs. For example, AIG(a, b) and AIG(b, a) should be merged together by the strash operation. You should implement a hash (…/util/myHash.h) and use it for this purpose. DO NOT use existing hashes such as "STL/hash_map" or "google/sparse_hash_map", etc. **A test program to test the implementation of your "myHash.h" will be included in the final grading.**

   You should perform "strash" operations only for the gates in the DFS list. For the gates that are not in the DFS list (i.e. those which cannot be reached from POs), please skip them. However, if any of the gates got merged (i.e. removed) during the "strash" operations, you should update the DFS list in the end.

5. When performing "merge" operation on a pair of gates, pick any of them to retain. That is, you can remove either one.

6. Boolean logic simulation is to explore the similarity of signals in the circuit netlist. If two signals have different simulation results, they are proven to be functionally distinct. However, if they react all the same for all simulation patterns, they are potentially equivalent and are called a "functionally

equivalent candidate (FEC) pair". Note that there may be more than two signals with the same simulation value. We call them a "FEC group" instead.

Alternatively, signals with completely inverse simulation values, although functionally non-equivalent, also imply functional similarity. In other words, one signal can be made equivalent to the other by adding an inverter on its output. In such case, we call them an "inverse functionally equivalent candidate (IFEC) pair" or "IFEC group".

In this project, only AIG and "const 0" gates will be included in the FEC pairs. PIs and POs won't be in any FEC pair, even if their values are the same as other gates.

You are free to choose any kind of simulation methodologies (e.g. event-driven, parallel pattern, etc), or even hybrid, in your implementation. Your goal is to efficiently distinguish as many functionally distinct signals as possible so that the remaining FEC/IFEC pairs are with higher chances to be equivalent.

7. Your simulation engine should be capable of generating random patterns as well as taking simulation patterns from files. The simulation pattern files are in the following format:

```
010100110010
...
```

Patterns are separated by one or more new lines or white space characters, and the length of each pattern should match the number of circuit inputs. The patterns, of course, should contain only '0' and '1' and no other character is allowed between the input assignments in a pattern. If there is an error in the input patterns, print out errors such as:

```
Error: Pattern(010100110010) length(12) does not match
the number of inputs(16) in a circuit!!
```

```
Error: Pattern(01010011x010) contains a non-0/1
character('x').
```

The leftmost character (0/1) in a pattern is corresponding to the assignment of the first PI in the PI list, and the rightest is corresponding to the last PI.

To test the correctness of the simulation results, you should provide an option (See `CIRSIMulate` command) to record the simulation results in a log file. The format for the log file is:

```
010100110010 1010
...
```

where the Boolean strings in each line represent the input patterns and the output response, respectively. There should be exactly ONE space character between them and NO space character at the beginning and end of the line.

8. When random simulation is performed, you should automatically determine how many patterns to simulate. Too few patterns may not be able to distinguish

functionally different signals, while too many patterns may result in significant runtime overhead. You should devise an efficient algorithm to simulate and collect the FEC pairs.

9. The signals in a FEC pair can be formally proven or disproven by a Boolean Satisfiability (SAT) solver. A public SAT solver (miniSat, http://minisat.se/) has been included in the *src/sat* directory. Please refer to the "README" file in the *src/sat* directory for building the interface between circuit and SAT engine. To use the SAT engine, you should include the header file "*sat.h*" in your code. In short, in order to call the SAT engine from circuit, you need to: (1) assign each gate with a distinct SAT variable ID and record the mapping between gates and these IDs, (2) call the interface function "*SatSolver::addAigCNF*" or "*SatSolver::addXorCNF*" to bulid the CNF formula for proof, (3) specify the proof target by the function "*SatSolver::assumeProperty"*. If necessary, release the previous proof target by the function "*SatSolver::assumeRelease"*, (4) prove it by "*SatSolver::assumeSolve*", and (5) obtain the counter-example (if satisfiable) by "*SatSolver::getValue*". Please refer to "*satTest.cpp*" in the "*sat/test*" directory for an example.

   Please note that SAT variable ID may be different from gate variable ID. The former will be used in SAT solver, and the latter is as defined in the aag file.

10. If two signals are proven functionally equivalent, merge them in the circuit netlist. That is, choose one representative signal and rewire the fanouts of other equivalent signals to this representative signal. The resultant circuit will then be simplified as a functionally reduced And-Inverter graph (FRAIG). However, if two signals are proven to be functionally different, the counter-example generated by the SAT engine can be a good test pattern to further distinguish signals in the FEC pairs. You should decide when to simulate this pattern in your FRAIG algorithm.

11. Optionally, you can implement a member function or provide a command to control the prove efforts of the SAT engine so that it will not stuck at the proof of a single FEC pair.

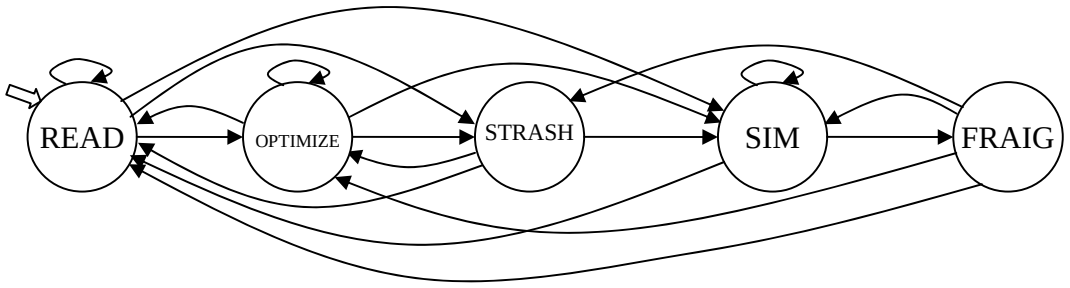
12. Most of the TODO's of this project are included in the *src/cir* directory. It contains a file *cirMgr.h* to define the circuit manager (i.e. `class CirMgr`) for circuit construction, reporting netlist, simulation and optimization (by FRAIG), etc. A global variable `CirMgr *cirMgr` as well as some member functions of `class CirMgr` should be defined in *cirMgr.cpp*. **You should use this global variable in circuit-related commands (defined in *cirCmd.cpp*) for all the circuit operations.** The AIG gate data structure and its constructing functions should be declared and defined in the files *cirGate.h* and *cirGate.cpp*. The optimization, simulation and fraig operations should be defined in the files *cirOpt.cpp*, *cirSim.cpp* and *cirFraig.cpp*, respectively.

13. You are free to define the data members and member functions for the classes `CirMgr` and `CirGate` (and maybe its inherited classes). However, please follow the naming convention in other homework/classes and be cautious on the memory and runtime efficiency.

14. PIs/POs/Const gates should remain intact throughout the program. Do not optimize them away by any command.

15. There should be flags to control the dependency of the circuit operations/ commands. In short, the circuit netlist should be constructed before any other circuit operations. The "strash" operation is operational, while the "simulation" must be performed before the "fraig" operation. "optimization" can be called whenever necessary. In addition, to get the best optimization result, you should allow the flow [optimization →] [strash →] simulation → fraig to perform multiple times. If the circuit is re-read from a file, *cirMgr* will be reset and all the netlist as well as the optimization, strash, simulation, and fraig results will be discarded.

    The dependency graph of the operations/commands is illustrated as follows:



    In short, the following command sequences are prohibited:

    (a) READ → FRAIG: simulation must be performed before fraig.

    (b) OPTIMIZE → FRAIG: simulation must be performed before fraig.

    (c) STRASH → FRAIG: simulation must be performed before fraig.

    (d) STRASH → STRASH: doesn't make sense to repeat strash

    (e) SIM → OPTIMIZE: don't optimize the circuit with FEC information

    (f) SIM → STRASH: don't optimize the circuit with FEC information

    (g) FRAIG → FRAIG: FEC pairs will be cleared after fraig

2. Supported Commands

In this project, you should support these new commands:

```
CIRRead:     read in a circuit and construct the netlist
CIRPrint:    print circuit
CIRGate:     report a gate
CIRWrite:    write the netlist to an ASCII AIG file (.aag)
CIRSWeep:    remove unused gates
CIROPTimize: perform trivial optimizations
CIRSTRash:   perform structural hash on the circuit netlist
CIRSIMulate: perform Boolean logic simulation on the circuit
CIRFraig:    perform FRAIG operation on the circuit
```

The first 4 commands are taken from Homework #6. If necessary, you can define your own commands for your own program testing or tuning. In that case, please explain those commands in your final report.

The reference program also provides one hidden commands:

```
CIRMiter:    create a miter circuit
```

and one extension to the `CIRWrite` command. You are welcome to use them from the reference program but don't need to implement them.

Please refer to Homework #3 and #4 for the lexicographic notations.

2.1 Command "CIRPrint"

Usage: **CIRPrint [-Summary | -Netlist | -PI | -PO | -FLoating | -FECpairs]**

Description: Report circuit information in different ways. If the circuit hasn't been constructed, print out an error message "`Error: circuit has not been read!!`". When "-Summary" is specified, print out the circuit statistics, and when "-Netlist" option is used, list all the gates in the topological (depth-first-search, DFS) order (Note: for the gates in DFS list only, those unreachable from POs won't be printed, even they are PIs). The "-PI", "-PO" and "-FLoating" options will report the lists of primary inputs (PIs), primary outputs (POs) and floating gates, respectively. The "-FECparis" reports the currently identified and unproven FEC pairs.

Example:

(Skipped: **-Summary | -Netlist | -PI | -PO | -FLoating**; please refer to HW#6)

fraig> cirp -fec

```
[0] 0 1296 1307 1465 1560 1579
[1] 1934 1938 1948 !1949 1951 !1952 1953
[2] 1777 !1883 !1885 1886 1887 1889 !1890 1891
```

The FEC pairs/groups are printed out one pair/group in a line. In each line, the variable IDs should be listed in ascending order. The '!' sign means this variable is inversely equivalent to other(s). Please make the first variable in the pair/group positive (i.e. no '!' sign). The first IDs in different lines do not need to be in any specific order.

2.3 Command "CIRGate"

Usage: **CIRGate <<(int gateId)> [<-FANIn | -FANOut><(int level)>]>**

Description: Report the gates in the circuit. If the "*gateId*" is not defined in the circuit, report "Error: Gate(*gateId*) not found!!". If the option "-FANIn" or "-FANOut" is NOT specified, print out the gate information such as ID, name (for PI and PO), gate type, and line number (as appeared in the AIGER file), FEC partners (i.e. the gates in the same FEC pair/group with IDs in ascending order), and simulation value (this may depend on your simulation mechanism). Otherwise, print out its fanin/fanout cone with the followed option (int level). Put proper indentations for different levels of fanin/fanout printing. If there is an inversion between a gate and its fanin/fanout, put a '!' before the fanin/fanout. If a gate whose fanin(s)/fanout(s) have been reported in previous lines, put a "(*)" after it is printed; do not repeatedly report its fanin(s)/fanout(s). If there is any undefined (i.e. floating) fanin, print "UNDEF" for its gate type.

Example:

fraig> cirg 25 // print out gate (25)

```
==================================================
= PO(25)"23GAT$PO", line 9                       =
= FECs:                                          =
= Value: 0100_1011_1100_1010_0000_1111_0101_0000 =
==================================================
```

fraig> cirg 13

```
==================================================
= AIG(13), line 12                               =
= FECs: 15 !30                                   =
= Value: 1100_1010_0001_0011_0000_1110_0110_0111 =
==================================================
```

(Skip the "-fanin/-fanout" options)

2.2 Command “CIRSWeep”

Usage: **CIRSWeep**

Description: Remove the gates that cannot be reached from POs.

Example:

For the example in the next page, after reading in the design, we have:

fraig> cirp

```
Circuit Statistics
==================
  PI           3
  PO           1
  AIG          6
------------------
  Total       10
```

Performing “CIRSWeep”:

fraig> cirsw

```
Sweeping: AIG(5) removed...
Sweeping: UNDEF(6) removed...
Sweeping: AIG(7) removed...
Sweeping: AIG(8) removed...
Sweeping: AIG(9) removed...
Sweeping: AIG(10) removed...
```

fraig> cirp

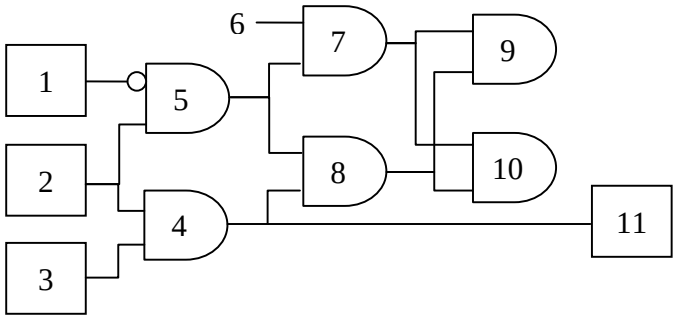
```
Circuit Statistics
==================
  PI           3
  PO           1
  AIG          1
------------------
  Total        5
```

fraig> cirp -n

```
[0] PI  2
[1] PI  3
[2] AIG 4 2 3
[3] PO  11 4
```

The netlist for the above example is:

```
aag 10 3 0 1 6
2
4
6
8
8 4 6
10 3 4
14 12 10
16 10 8
18 14 16
20 14 16
```



2.3 Command “CIROPTimize”

Usage: **CIROPTimize**

Description: Perform trivial circuit optimizations including constant propagation and redundant gate removal. This command can be evoked anytime except when “CIRSIMulate” command is just called. In such case, an error message “`Error: circuit has been simulated!! Do "CIRFraig" first!!`” will be issued.

Example:

fraig> ciropt

2.3 Command “CIRSTRash”

Usage: **CIRSTRash**

Description: Merge the structurally equivalent gates. After the merger, the fanouts of the merged gate will be re-connected to the gate that merges it. Unless the circuit is re-read, or optimization or “fraig” operation has been performed, it does not make sense to perform strash multiple times. If repeated “CIRSTRash” is issued, output an error message: “`Error: strash operation has already been performed!!`”.

Example:

fraig> cirstrash

2.4 Command “CIRSimulate”

Usage: **CIRSimulate <-Random | -File <string patternFile>>**

[-Output (string logFile)]

Description: Perform circuit simulation to distinguish the functionally different signals and thus collect the FEC pairs/groups. If “-Random” option is specified, perform random simulation until *you think* it is time to stop. You should devise the stopping criteria for the random simulation so that

the FEC identification can be as efficient as possible. On the other hand, if the option “-File” is used, read the simulation patterns from the pattern file. The “-Output” option designates the name of the log file. Please refer to Section 1 for the input and output pattern formats. After the simulations, report the number of input patterns that have been simulated.

Example:

fraig> cirsim -random // perform random simulation

```
1024 patterns simulated.
```

fraig> cirsim -file in.100 -o 100.log // specify the input/output files

```
100 patterns simulated.
```

Note:

If there is an UNDEF gate in the fanin cones of POs, set this UNDEF gate to value 0 for simulation. However, we will NOT test this kind of case for the CIRSIMulate and CIRFraig commands.

2.5 Command “CIRFraig”

Usage: **CIRFraig**

Description: Based on the identified (I)FEC pairs/groups, perform fraig operations on the circuit. All the SAT-proven equivalent gates should be merged together, while it is optional (to you) to re-simulate the witness patterns of the non-equivalent FEC pairs.

Example:

fraig> cirfraig // perform fraig operation

2.6 Command “CIRWrite”

Usage: **CIRWrite [(int gateId)] [-Output (string aagFile)]**

Description: (Skipped) In addition to writing out the entire circuit, you can optionally write out a cone of sub-circuit rooted on gate “gateId”. The output, in such case, will be a single-output circuit.

Example:

fraig> cirwrite 38 -o 38.aag // write the fanin cone of gate 38

2.7 Command “CIRMiter”

Usage: **CIRMiter <(string inFile1)> <(string inFile2)>**

Description: Construct a miter from the circuits “inFile1” and “inFile2”. The number of PIs (POs) from both circuits must be the same. A temporary circuit description file will be stored in “/tmp/miter.aag” and read in at the end of this command.

3. What you should do?

You are encouraged to follow the steps below for this homework assignment:

1. Read the specification carefully and make sure you understand the requirements.

2. Think first how you are going to write the program, assuming you don’t have the reference code.

3. Type “make 32” or “make 64” to switch to a proper platform. The default is 64-bit platform.

4. Please be advised that a good design of the data structure is very important. Please be thoughtful in designing *cirGate.h* and *cirMgr.h*. However, you should compile and test your code whenever applicable. Do not write a bunch of codes without compiling and testing. In addition, you should revise and modularize the code from time to time. Remember, a “good-looking” code implies an “easier-to-maintain” one, which is the key to succeed in this final project.

5. For the circuit parsing part of codes in HW#6, it is OK to use other student’s codes if you didn’t do a good job on HW#6. We will not treat it plagiarism on those codes. However, DO NOT copy and paste other parts of the header files or source codes.

6. Implement “CIRSWeep” command first. Remember to test “CIRPrint” and “CIRGate” before and after this command.

7. Implement “CIROPTimize” command. Please note that although there are only 4 cases, the codes can be messy if you don’t think carefully before coding. Make sure the circuit functionality remain unchanged after this command.

8. Complete “*myHash.h*” in *src/util* and then implement the “CIRSTRash” command. Be careful in merging the circuit netlist. Use “CIRPrint” and “CIRGate” to test if you have correctly reconstructed the circuit.

9. All the features up to this point are more “friendly”. You should try to accomplish and test them in time. The following features can be quite challenging. Please make sure you spare enough time. Otherwise, it can be fruitless.

10. Implement “CIRSIMulate” command. While the “simulation” function itself may be easy, how to efficiently collect the FEC pairs can be quite challenging. In addition, you should control the effort for the random simulation. Distinguishing too many FEC pairs may significantly increasing the simulation time, while keeping the simulation effort low may then result in long FRAIG runtime and bad optimization result.

11. Get familiar with the SAT engine interface. You can refer to the example in *src/sat/test*.

12. Implement “CIRFraig” command. Please make sure you understand the concepts. Be smart in calling the SAT engine and using its result to simplify the circuit as well as distinguish the remaining FEC pairs. Please note that some of the cases in *tests.fraig* can be simplified to a constant “0” or “1” node. Please also pay attention to the program performance. It will be included in the grading (a small factor though).

13. Write a professional report. A “cover” with your name, school ID, and valid contact information is a MUST as we may need to contact you if we have any question on your final project. Inside your report, please include your design of the data structure, your algorithms at least for simulation and FRAIG, and the results and analysis of the experiments. Comparison to the reference program is welcome, but please discuss the performance and bottleneck (if any) of your program. In addition, **comments to the class** are also very welcome. Name your report file as <studentID>.pdf.

14. If the SelfCheck program fails, check the line (line 40):

    ```
    @temp = split /_hw/, $arg;
    ```

    Change it to:

    ```
    @temp = split /_fraig/, $arg;
    ```

4. Grading

The final project is weighted 40% of your term grade. Its score will be (roughly) based on the following criteria:

Report	15%	(Please do spend time to write a good report!!)
Implementation	85%	
- Sweeping (10%)		(correctness: 90%, performance: 10%)
- Optimization (15%)		(correctness: 90%, performance: 10%)
- Strash (15%)		(correctness: 90%, performance: 10%)
- Simulation (20%)		(correctness: 70%, performance: 30%)

- Fraig (25%) (correctness: 50%, performance: 50%)

We will test your submitted programs with various testcases and compare the outputs with our reference program. Minor differences due to printing alignment, spacing, error message, etc can be tolerated. The debugging (circuit restructuring) message in various optimization commands can be ignored. However, to assist TAs for easier grading work, please try to match your output with ours (especially the printed order of gates).

The testing results of your final project will be announced in a couple of days after the due date. Please pay attention to the announcement and e-mails. No correction on the final grade will be possible after it is submitted to the registration office.