Final Project

FRAIG:

Functionally Reduced And-Inverter Graph

資料結構與程式設計

Data Structure and Programming

12/11/2019

1

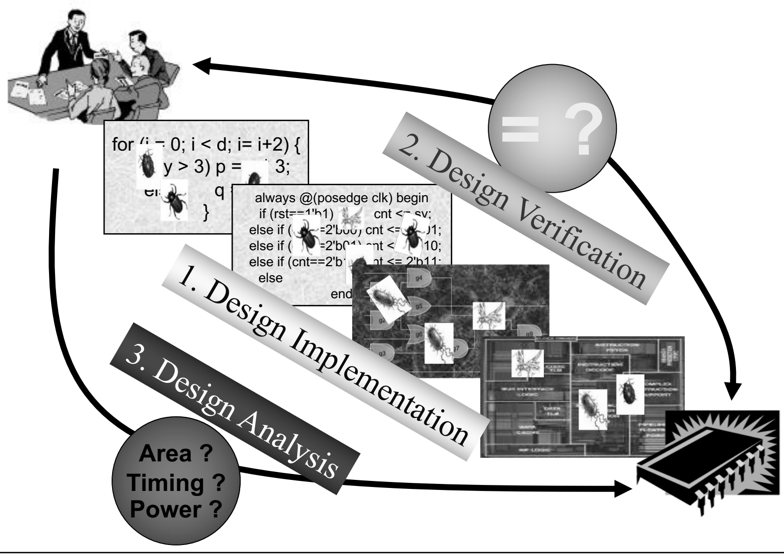
Functionally – Reduced – AIG (FRAIG)

- AIG: you have learned it in HW#6
- Functionally?
  - Well, AIG represents a circuit, so it represents some Boolean function.
- Reduced?
  - Reduction on AIG ➔ Simplifying graph
  - How to simplify AIG?
- Functionally Reduced?
  - (e.g.) Two functionally equivalent nodes can be merged together
  - (e.g.) Simplify circuit by constant propagation

2

Electronic Design Automation
(EDA)

How is a “chip” designed nowadays?



How/What to optimize a circuit?

- ◆ Area
  - Reduce the number of gates
  - Moreover, using library cells of smaller sizes
    ➔ but they will have weaker driving capability
- ◆ Timing
  - Shorten the longest path
  - Additionally, insert buffers and/or enlarge the cells to increase the driving capability
- ◆ Power
  - Reduce the switching activities
  - Moreover, shutdown the sub-circuit that is not currently used

Optimization trade-offs

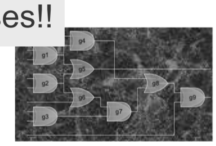
- ◆ In general, area, timing, and power optimizations contradict with each other
- ◆ Moreover, different stages of design flow have different granularities and complexities for circuit optimization
  - HDL (e.g. Verilog) // algorithm
  - Gate (Boolean) // logic
  - Layout (transistor) // RC network

A simplified view of circuit optimization

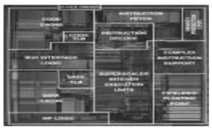
- ◆HDL (Verilog)
  - Architectural and algorithmic optimizations

```
always @(posedge clk) begin
  if (rst==1'b1)          cnt <= sv;
  else if (cnt==2'b00) cnt <= 2'b01;
  else if (cnt==2'b01) cnt <= 2'b10;
  else if (cnt==2'b10) cnt <= 2'b11;
  else                    cnt <= sv;
end
```

- ◆Gate (Boolean) What FRAIG focuses!!
  - Minimize gate counts under reasonable timing and power constraints



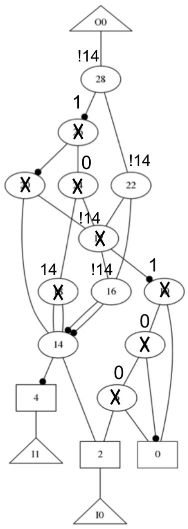
- ◆Layout (transistor)
  - Minimize wire length for timing and power optimizations with limited area overhead

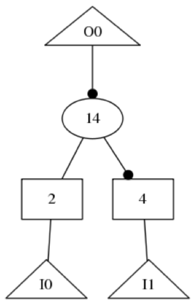


A simple example

- ◆sim05.aag

original:
12 AIGs





optimized:
1 AIG

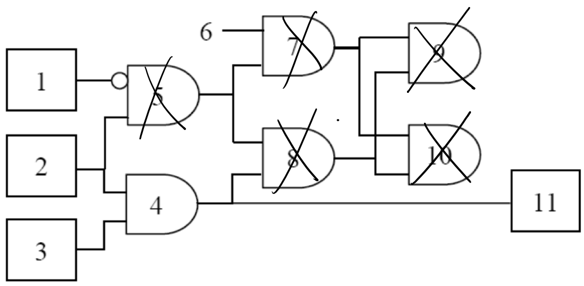
Functionally Reduced AIG

In the final project, we will apply 4 different types of optimization techniqeus:

1. Unused gate sweeping
2. Trivial optimization (constant propagation)
3. Simplification by structural hash
4. FRAIG: Equivalence gate merging

Unused Gate Sweeping

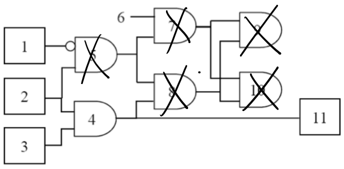
◆ Sweeping out those gates that are not reachable from POs.



Example: opt07.aag

Unused Gate Sweeping

- Command: CIRSWeep
  - Can be called whenever necessary.
  - Note: do not remove unused PIs.
  - After this command, all gates except for the unused PIs will be in the DFS list.
  - Note: be sure to update the reporting for "CIRPrint -FLoating".
- In the previous example (cirp -fl):
  - Before:
    - Defined but not used: 9 10
    - Gates with floating fanin: 7
  - After:
    - Defined but not used: 1

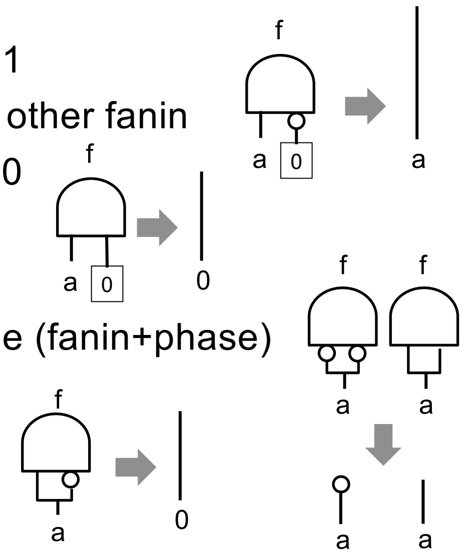


Functionally Reduced AIG

1. Unused gate sweeping
2. Trivial optimization (constant propagation)
3. Simplification by structural hash
4. FRAIG: Equivalence gate merging

Trivial optimization

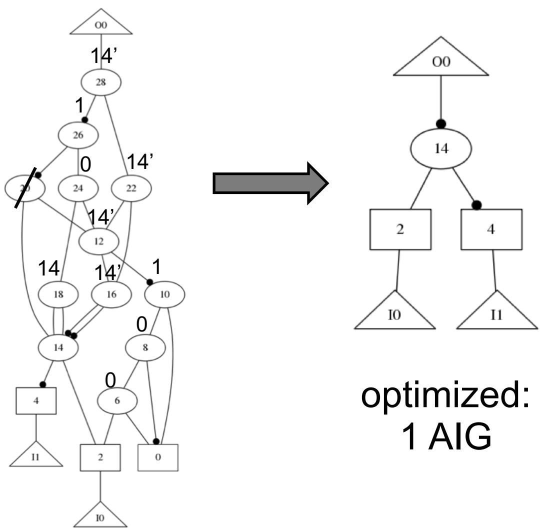
1. Fanin has constant 1
   ➔ Replaced by the other fanin
2. Fanin has constant 0
   ➔ Replaced with 0
3. Identical fanins
   ➔ Replaced with the (fanin+phase)
4. Inverted fanins
   ➔ Replaced with 0



A simple example

◆sim05.aag

original:
12 AIGs



optimized:
1 AIG

Trivial optimization

- Command: CIROPTimize
  - Can be called whenever necessary
  - Scan the DFS list and perform optimization ONCE. Don't repeatedly optimize the circuit. ➔ The latter can be achieved by calling CIROPTimize multiple times.
  - Don't perform optimization during CIRRead
- Do not remove PIs / POs
- Some UNDEF or defined-but-not-used gates may disappear!
- Some gates (with side input = constant 0) may become "defined-but-not-used".

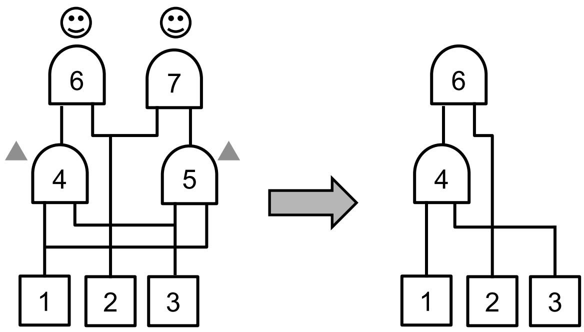
Functionally Reduced AIG

1. Unused gate sweeping
2. Trivial optimization (constant propagation)
3. Simplification by structural hash
4. FRAIG: Equivalence gate merging

f1 = AND(a, b) f2 = AND(b, a)

Structural Hash (Strash)

- Example:



Structural Hash (Strash)

- Problem: How to identify two AIG gates in a circuit that have the same inputs?
  - [Method 1] Check for O(n^2) pairs of gates
  - [Method 2] For each gate, check its fanouts
    - How many checks?
  - [Method 3] For each gate, create hash table <fanins, this gate>
    - How many checks?
- We will pick method 3 in our project
  - You can modify your “util/myHashSet.h” to HashMap, or use stl/unordered_map
- Although it is possible to perform strash during circuit parsing, we choose to make “strash” a separate command.
  ➔ CIRSTRash
- Note: Order matters!! You should merge from PIs to POs (Why??)

Structural Hash Algorithm

◆ HashMap<HashKey, HashData> hash;
- HashKey depends on gate type & list of fanins
- HashData is Gate*

➔ What if we have only AIG?
➔ How about inverted match?

◆ class HashKey
```
{
   size_t operator () () const { // as hash function }
   bool operator == (const HashKey& k) const {…}
private:
  ~~Gate *g0, *g1;~~  size_t  in0, in1;
};                    // use LSB for inverted phase
```
◆ HashData can be size_t
◆ For unordered_map, need to define "hash" class

Structural Hash Algorithm

◆for_each_gate_from_pi_to_po(gate, hash)
```
    // Create the hash key by gate's fanins
    HashKey<...> k(...);  // a function of fanins
    size_t  mergeGate;
    if (hash.check(k, mergeGate) == true)
       // mergeGate is set when found
       mergeGate.merge(gate);
    else  hash.forceInsert(k, gate);
```
◆size_t ? ➔ CirGateV
Create a wrapper class on top of a size_t !!

Notes about CIRSTRash

◆Perform strash only on gates in DFS list
- Do not perform strash on gates which cannot be reached from POs
- This is to avoid those unreachable gates appearing in DFS list

◆It doesn't make sense to perform strash again before doing other optimizations
- CIRSTRash cannot be repeated called

Maintaining Netlist Consistency

◆Once circuit is simplified, some gates may become invalid.
- How to maintain the netlist consistency?

1. Properly re-connect fanins/fanouts
2. Properly release memory (if necessary)
3. Properly update the lists in CirMgr (Note: PI/PO lists should never be changed)

Functionally Reduced AIG

1. Unused gate sweeping
2. Trivial optimization
3. Simplification by structural hash
4. FRAIG: Equivalence gate merging

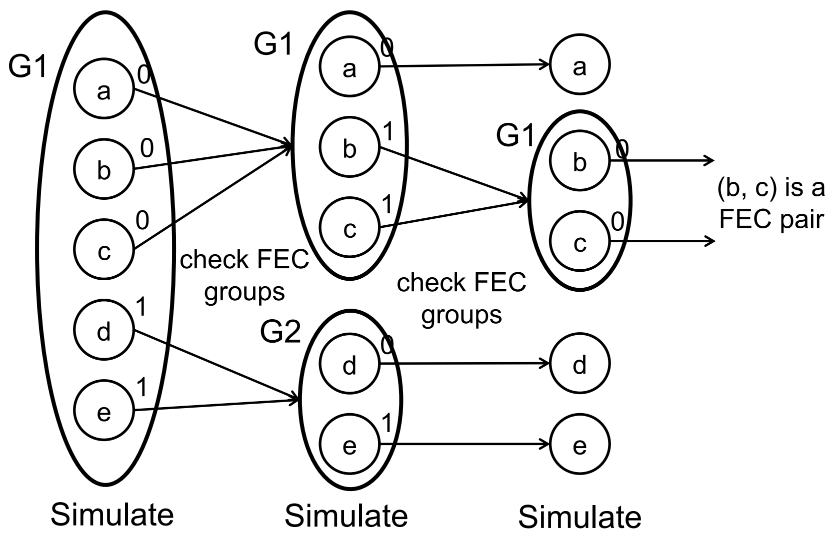
FRAIG: Merging equivalent gates

- Some gates are NOT structurally equivalent, but functionally equivalent.
  - Cannot be detected by strash
  - e.g. $ab + c \equiv (a + c)(b + c)$
- How to know two gates are functionally equivalent?
  - By simulation? (If two gates have the same value)
  - ➔ Not quite possible, equivalence requires to enumerate "ALL input patterns" // exhaustive simulation
  - Need "formal (mathematical) proof"!!
  - ➔ But, what to prove? $O(n^2)$ pairs?
  - ➔ By simulation!! // to check the potential equivalence

FEC Pairs

- Functionally Equivalent Candidate (FEC)
  - For all simulated patterns, if two signals always have the same response, they are very likely to be equivalent.
- Properties
  - Two signals can be separated if they have different simulation responses for at least ONE input pattern
  - Two paired signals can be separated by simulation, but two separated signals won't get paired again
    - Singleton signal won't be in any FEC pair anymore

Identify FECs by Simulation



Simulation Algorithm

◆ All-gate simulation:
Perform simulation for each gate on the DFS list
- void CirMgr::simulate() {
  for_each_gate(gate, _dfsList) gate->simulate(); }

◆ Event-driven simulation:
Perform simulation only if any of the fanins changes value
- void CirMgr::simulate() {
  for_each_PO(po, _dfsList) po->simulate(); }
  bool CirAigGate::simulate() {
  Recursively simulate each fanin.
  If (no fanin has value change) return false;
  Simulate this gate;
  if (value changed) return true;
  return false;
  }

Discussions: Simulation algorithm trade-offs

◆ All-gate simulation or event-driven?

◆ Evaluation
- By operator? By if-else? By table lookup?

◆ To detect FEC pairs, how many simulation patterns are enough?
- Stop if no new FEC pair is found?
- (Dynamically) Controlled by “#failTimes”

◆ Patterns
- Single pattern? Parallel pattern?

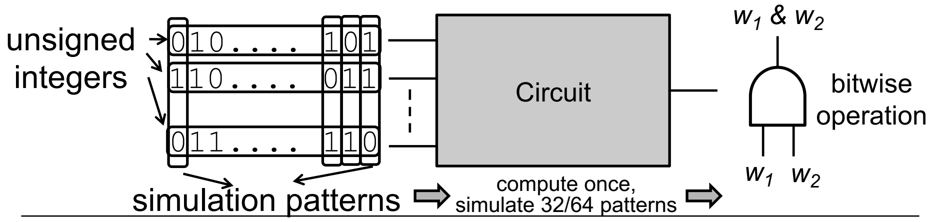
Parallel-Pattern Simulation for FEC Identification

◆Note: The speed overhead in bitwise operations is very small.

- Most of the programming languages (e.g. C/C++) support "bit-wise" operations (e.g. &, |, ~ in C/C++).

◆Idea

- Using 32- or 64-bit unsigned integer to pack 32 or 64 patterns into a word



How many patterns to parallelize?

◆In practice, max parallelization will lead to the best simulation performance

- Use the max "unsigned int" to store the parallel patterns (e.g. size_t in C/C++)

[Discussion]

◆Can we go beyond 32/64 bits?

- e.g. 1024-bit

◆What are the pros and cons?

◆How about the FEC detection rate?

Identify FECs by Simulation

1. Initial: put all the signals in ONE FEC group.
2. Add this FEC group into fecGrps (list of FEC groups)
3. Randomly simulate the entire circuit
4. for_each(fecGrp, fecGrps):

```
Hash<SimValue, FECGroup> newFecGrps;
for_each(gate, fecGrp)
   grp = newFecGrps.check(gate);
   if (grp != 0)   // existed
      grp.add(gate);
   else  newFecGrps.add(createNewGroup(gate));
CollectValidFecGrp(newFecGrps, fecGrp, fecGrps);
```

5. Repeat 3-4 until no new FEC Group can be identified, or efforts exceed certain limit.

Mass simulation → Identify FEC pairs

How to prove/disprove
the equivalence of gates in an FEC pair?

Convert it into a SAT problem!!!

Boolean Satisfiability (SAT) Problem

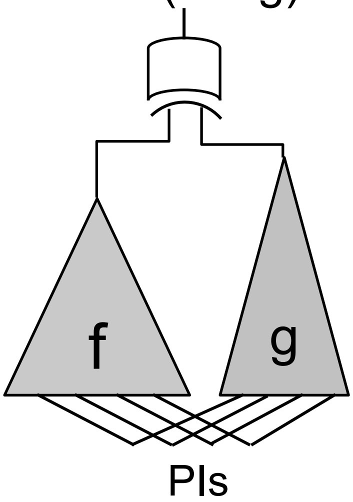
- ◆Given a Boolean function f(X), find an input assignment X = A such that f(A) = 1.
  - Satisfiable: if such an assignment is found
  - Unsatisfiable: if no assignment is possible
    - i.e. All assignments make f(X) = 0
  - Undecided: can't find a satisfying assignment, but haven't exhaust the search
  - SAT Game: https://goo.gl/9JJVmJ
- ◆Complexity?
  - First proven NP-complete problem by Dr. S. Cook in 1971 (Turing Award winner)

How to prove the equivalence of FEC gates?

- ◆In general, given two Boolean functions, f, g, how to check if they are equivalent?
- ◆Note:
  - SAT proves things by contraposition
  - ➔ By showing that it is *impossible* to find an assignment to make f != g.
  - ➔ Create a SAT problem F ≡ (f != g), showing that it is unsatisfiable.
  - ➔ Note: f != g ➔ an XOR gate

How to prove the equivalence of FEC gates?

$F \equiv (f \,!\!= g)$



(f, g) is an FEC pair
➔ SAT? (F = 1)

If UNSAT ➔ f = g

If SAT ➔ f != g with an input assignment A that can distinguish (f, g) and potentially distinguish other pairs

FRAIG flow

1. Simulation
   a) Put all signals in the same group
   b) Simulate the circuit. If two signals have different simulation results, separate them into different groups
   c) Repeat (b) until no more signals can be distinguished, or the simulation efforts exceed a preset limit
   d) Collect the functionally equivalent candidate (FEC) pairs

FRAIG flow

2. For each FEC pair, call Boolean Satisfiability (SAT) engine to prove their equivalence
    1. If they are equivalent, merge them together ➔ remove one of them
    2. If they are NOT equivalent, acquire the counter-example (CEX) that distinguishes them
    3. Repeat until all the FEC pairs have been proved, or enough CEXes (2.2) have been collected ➔ Repeat "1. Simulation"

In short...

1. Simulation identifies a group of FEC pairs
2. For each FEC pair, say (f, g), call SAT engine to check if (f != g) is satisfiable
3. If UNSAT ➔ f = g ➔ f can replace g
4. If SAT
   ➔ collect the pattern that witness (f != g)
   ➔ simulate again to see if it can distinguish other FEC pairs
5. Repeat 2 ~ 4

➔ So the remaining problems are: How to call SAT engine? How to create SAT proof instance?

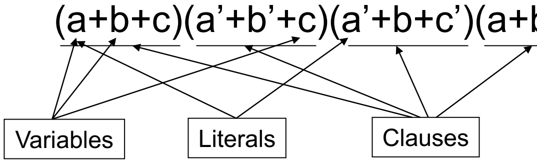
Boolean Satisfiability (SAT) Engine

- ◆An engine (i.e. a program/library/function) that can prove or disprove a Boolean Satisfiability problem
  - Called a "SAT engine" or "SAT solver"
- ◆A well-studied CS problem, but was once generally thought as an intractable problem.
  - Many practical, powerful, and brilliant ideas were brought up by EDA researchers in early 2000 ➔ Orders of improvement
  - ➔ Made a revolutionary change on the applications of SAT

Creating Proof Instance

- ◆Proof instance: the formula under proof
- ◆Conjunctive Normal Form (CNF)
  - Most modern SAT engines represent the proof instances in CNF
  - Actually a "product of sum" representation

$$(a+b+c)(a'+b'+c)(a'+b+c')(a+b'+c')$$



  - To be satisfied, all the clauses need to be 1

Converting circuit to CNF

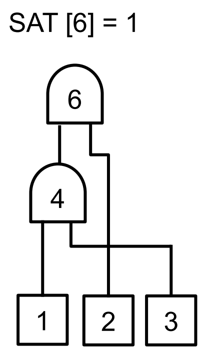
◆ Each gate is assigned a variable ID
◆ Each gate is converted to a set of CNF clauses based on its fanin variables
- g = AND(a, b)

1. a = 0 → g = 0 (a + !g)
2. b = 0 → g = 0 (b + !g)
3. a = 1 & b = 1 → g = 1 (!a + !b + g)

◆ To solve (f = 1), add a (f) clause
- SAT engine is to check if all the clauses can be satisfiable at the same time.

Converting circuit to CNF

◆ Example:



(1 + !4)(3 + !4)(!1 + !3 + 4)
(4 + !6)(2 + !6)(!2 + !4 + 6)
(6)

Calling SAT engine

- Create a solver object
- Add clauses → proof instance
- (optional) Set proof limits
- Solve()!!

→ We provide a SAT interface in "sat.h"

- (FYI) Incremental SAT
  - Reuse the partial learned information

Using SAT to prove FEC pair

1. Create a solver object
   ```
   SatSolver solver;
   solver.initialize();
   ```
2. Create CNF for the circuit
   - For each gate in the circuit, create a variable for it
     - `solver.newVar();`
   - For each gate in the circuit, create CNF clauses for it
     - `solver.addAigCNF(v, v1, ph1, v2, ph2);`
   - Remember to take care of CONST gate
3. Create the proof instance for $F \equiv (f != g)$
   - Add clauses for F
     - **solver.addXorCNF(FVar, fVar, fPh, gVar, gPh);**
   - Call SAT to prove
     - `solver.assumeRelease();`
     - `solver.assumeProperty(newV, true);`
     - `bool isSat = solver.assumpSolve();`
     - `getSatAssignment(solver, patterns);`

Notes about FEC proof

- Order matters!!
  - Proving from PIs to POs can greatly reduce the proof effort
  - DFS or BFS?
- Don't waste SAT-generated patterns (for f != g)
  - Pack them for parallel pattern simulation
- Many FEC pairs are actually (f, 1) or (f, 0).
  - Should we do anything special for them?
- It's OK to skip some proofs. (Why?)
  - Skip it or limit the proof effort (e.g. #conflicts)
- Incremental SAT
- Balance between simulation and proof efforts

Some advices

- Please do not fall into 軍備競賽...
  - Although it is possible you can implement a version that is 10X faster than mine...
- It's OK that you CANNOT finish the project.
  - I don't expect many people to finish the project.
  - Think: 你的電子學有拿 100 分嗎?
- Please DO NOT spend 80% time on 20% points
  - e.g. parser error message, circuit optimization
- Always keep your code simple and straight!!
  - Always modularize your code
  - Compile and test from time to time

References

- Functionally Reduced And-Inverter-Graph
  - http://www.eecs.berkeley.edu/~alanmi/publications/2005/tech05_fraigs.pdf
- A System for Sequential Synthesis and Verification
  - http://www.eecs.berkeley.edu/~alanmi/abc/
- SAT solver
  - http://www.satcompetition.org/
  - http://www.princeton.edu/~chaff/publication/DAC2001v56.pdf
  - http://www.princeton.edu/~chaff/publication/cade_cav_2002.pdf