Satisfiability Machine Design
Overview

- Purpose of project
- Explanation of Satisfiability
- The need for satisfiability solutions
- Software satisfiability algorithms
- Reconfigurable satisfiability hardware
- Results of project
Purpose of Project

❖ To develop software to generate VHDL for each problem. This VHDL creates a very fast satisfiability solver in a FPGA which can then be used to solve the problem.

❖ Why program it for each problem?
  • Too much hardware is needed for a large static problem solver
  • Takes advantage of most important aspect of FPGA -- Reconfigurable!
The Satisfiability Problem

Given:

- a set of \( n \) Boolean variables \( x_1, x_2, \ldots, x_n \)
- a set of literals, where a literal is a variable \( x_i \) or the complement of a variable \( x'_i \)
- a set of distinctive clauses \( C_1 \ldots C_m \) where each clause consists of literals combined by OR
- The function \( CNF = C_1 \cdot C_2 \cdot \ldots \cdot C_m \) CNF = conjunctive normal form

We say the equation is **satisfiable** if there exists a combination of truth values of \( x \)’s that make the equation \( F \) true.

If no such solution exists, it is **unsatisfiable**
The Satisfiability Problem (cont.)

- Not an easy problem to solve!
- The first NP-complete problem to be found:
  - NP = non-deterministic polynomial.
  - Complete means it is the key to a set of problems.
  - Has never been proven that there is no easy solution! (i.e., non-exponential solution)
  - Another is graph coloring
An exact solution calculation increases exponentially with the increasing number of variables and clauses.

An average satisfiability problem may have 50 inputs, 100 clauses and 300 literals.

A large problem can have 1000 inputs, 3000 clauses and 10000 literals!
Why do we need satisfiability?

- By merging all equations into large equation, a system of Boolean equations can be solved using satisfiability.
- Large systems of equations are used in important CAD problems such as:
  - timing verification
  - layout calculation
  - routing analysis
- State assignment & minimization
- Automatic test-pattern generation
- Automatic theorem-proving
Solving in Software (brief)

- No algorithm has been found that is non-exponential for all cases.
- There are many algorithms for solving satisfiability that are efficient for most cases.

- Two classes:
  - Exhaustive search, exponentially complexity
  - Backtracking algorithms, worst case exponential usually better
    - First Davis-Putnam algorithm -- 1960’s
    - GRASP (Generic seaRch Algorithm for the Satisfiability Problem)
      - tree pruning & bookkeeping backtracking algorithm 1996
Satisfiability in Hardware

- **Static Hardware** (only programmed once)
  - 1997 ESOP Minimization/Satisfiability machine
  - Doesn’t require programming for each case
  - Is VERY limited on number of variables
  - Fast calculation
Satisfiability in Hardware

- **Reconfigurable Hardware**
  - Requires VHDL generation, place & route and programming time for each problem
  - Hardware is not too complex and can do very large problems
  - Very fast calculation
Reconfigurable machines

- Use efficient satisfiability **backtracking algorithms** in hardware.
- For each problem **unique** VHDL must be generated and programmed in FPGA that implements the algorithm.
- Because most (except ESOP) satisfiability problems **do not use all variables and terms**, hardware does not grow exponentially with number of variables and clauses.
- Can have a speedup of 1000x over the best software algorithm.
- The only tradeoff is the time to generate VHDL and program the FPGA for each problem.
Each variable in Canonical Normal Form (CNF) has its own state machine.

Data path is the combinational equation of all the variables representing the output function.

The global controller sets up the system and keeps track of when it is complete.
Fig. 1. Block diagram for the basic architecture (CE), consisting of an array of FSMs (#1 ... #n), a datapath, and a global controller. The variables $x_i$ and the CNF are modeled in 3-valued logic.
State Machine for each variable

One State Machine block

FSM

CNF

FT

TT

CNF

FB

FB

Varout

One State Machine block
Fig. 2. State diagram for an FSM of the architecture CE. The inputs are FT (from top) and FB (from bottom) that activate the FSM, and the 3-valued CNF. The output signals displayed inside the states are the variable value, and the signals TT (to top) and TB (to bottom) that activate the previous and next FSM.
Simple Example

CNF = (x+y)(y’+z’)(z+x)

Assign 0 to variable
LOOP(
  • If CNF = 1 we found a solution, end
  • If CNF = 0 we made the function unsatisfiable. Set variable to 1. Continue loop.
  • If CNF = X activate the next FSM down.
  • If we’ve tried both 1 & 0 assign variable to X and activate next higher FSM
  ) END LOOP

Solution Found!

If this ever becomes 1, we have proved unsatisfiability
Project Results

- VHDL for state machine
- VHDL code to implement 3 valued logic
- C code to generate VHDL for solver logic
VHDL for single state machine

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.ternary.ALL;

ENTITY  satis_control IS
  PORT (  
    clk : IN STD_LOGIC;  -- input clock
    reset_n : IN STD_LOGIC;  -- asynchronous reset
    from_top : IN STD_LOGIC;  -- signal from higher FSM
    from_bot : IN STD_LOGIC;  -- signal from lower FSM
    to_top : BUFFER STD_LOGIC;  -- signal to higher FSM
    to_bot : BUFFER STD_LOGIC;  -- signal to lower FSM
    Fin : INTERNARY;  -- signal in
    Vout : BUFFER TERNARY;  -- signal out
  );
END satis_control;

ARCHITECTURE structural OF  satis_control IS
  -- Define all the states of the state machine.
  TYPE state_type IS ( inactive, active_low, active_high, activate_bot_low, activate_bot_high, tried_both );
  SIGNAL present_state, next_state : state_type;

  BEGIN
    update : process ( reset_n, clk, next_state )
    BEGIN
      -- process to update the present state
      IF ( reset_n = '0' ) THEN
        present_state <= inactive;
      ELSEIF ( clk'EVENT AND clk = '1' ) THEN
        present_state <= next_state;
      END IF;
    END PROCESS;

    transitions : process ( present_state, from_top, from_bot, Fin )
    BEGIN
      -- process to calculate the next state and its associated outputs.
      CASE present_state IS
        WHEN inactive =>$
          Vout <= '1': to_top <= '1': to_bot <= '0': Vout <= '0':
          IF ( Fin = '1' ) THEN
            next_state <= activate_bot_low;
          ELSIF ( Fin = '0' ) THEN
            next_state <= active_high;
          ELSE
            next_state <= inactive;
          END IF;
        WHEN active_high =>$
          Vout <= '0': to_top <= '0': to_bot <= '0': Vout <= '1':
          IF ( Fin = '1' ) THEN
            next_state <= activate_bot_high;
          ELSIF ( Fin = '0' ) THEN
            next_state <= tried_both;
          ELSE
            next_state <= inactive;
          END IF;
        WHEN activate_bot_low =>$
          Vout <= '0': to_top <= '0': to_bot <= '1': Vout <= '1':
          IF ( from_bot = '1' ) THEN
            next_state <= active_high;
          ELSE
            next_state <= activate_bot_low;
          END IF;
        WHEN activate_bot_high =>$;
          Vout <= '0': to_top <= '0': to_bot <= '0': Vout <= '1':
          IF ( from_bot = '1' ) THEN
            next_state <= tried_both;
          ELSE
            next_state <= activate_bot_high;
          END IF;
        WHEN tried_both =>$;
          Vout <= '0': to_top <= '0': to_bot <= '0': Vout <= '1':
          IF ( from_top = '1' ) THEN
            next_state <= active_low;
          ELSE
            next_state <= inactive;
          END IF;
      end case;
    END PROCESS;
  END structural;
```
VHDL for 3 valued logic

- Overloads operators so “+” “*” and NOT can be used for TERNARY signals.
Program will parse a CNF format file and generate the entity and architecture for the solver.

A ‘c’ on first of line denotes a comment a p defines the file parameters

- p (file format) (# vars)(# clauses)
C code to generate VHDL for logic

Program begins by opening the CNF file & copying a template file (contains the entity description) to the output VHDL file
C code to generate VHDL for logic

Next the code reads a line and parses the preamble line to find out the number of variables and clauses.
C code to generate VHDL for logic

The rest of the preamble is parsed and the signals between FSMs are defined for the number of variables.

From_top, to_top & var_out
C code to generate VHDL for logic

The outer for loop goes through each line, the inner through each character of the line.

It generates +, * and NOTs and strings them together

i.e 1 2 0 would become (var_out(1) + var_out(2)) *

The last few statements generate the state machines and connect them together
A simple example \( F = (x+y)(y'+z')(z+x) \)

The above CNF file generates the VHDL code to the right.
Schematic after synthesis

Datapath
Logic

FF Next State
Logic

FFs
Timing Diagram for simple problem

It solves the problem 10 cycles after it is started

Fout = 1, solution found
A more complex example: hole6 of the DIMACS benchmark suite

```
c File: hole6.cnf

SOURCE: John Hooker (jh38+@andrew.cmu.edu)

DESCRIPTION: Pigeon hole problem of placing n (for file holn) pigeons
in n+1 holes without placing 2 pigeons in the same hole

NOTE: Part of the collection at the Forschungsinstitut fuer
anwendungsorientierte Wissensverarbeitung in Ulm Germany.

NOTE: Not satisfiable

```

42 variables & 133 clauses!

Schematic is 4 pages long!
Mapping report for hole6

- Max speed is 2MHz, with optimization could be higher
- Uses 1120 FG and 251 FFs

Total accumulated area:
- Number of BUFG: 1
- Number of CLB Flip Flops: 251
- Number of FG Function Generators: 1120
- Number of H Function Generators: 134
- Number of IBUF: 2
- Number of OBUF: 2
- Number of Packed CLBs: 476
- Number of STARTUP: 1

Device Utilization for 4013ePQ160

<table>
<thead>
<tr>
<th>Resource</th>
<th>Used</th>
<th>Avail</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOs</td>
<td>5</td>
<td>129</td>
<td>3.88%</td>
</tr>
<tr>
<td>FG Function Generators</td>
<td>1120</td>
<td>1152</td>
<td>97.22%</td>
</tr>
<tr>
<td>H Function Generators</td>
<td>134</td>
<td>576</td>
<td>23.26%</td>
</tr>
<tr>
<td>CLB Flip Flops</td>
<td>251</td>
<td>1152</td>
<td>21.79%</td>
</tr>
<tr>
<td>Clock</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>: Frequency</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clk</td>
<td></td>
<td></td>
<td>2.0 MHz</td>
</tr>
</tbody>
</table>
To compare, I created the VHDL and simulated the design for increasing size hole benchmark problems until it solved (or proved) the satisfiability of the problem.

Below is the results:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>inputs</th>
<th>clauses</th>
<th>literals</th>
<th>Clocks</th>
<th>time @10MHz</th>
<th>GRASP time on 300Mhz Pent SU</th>
<th>Cost(FGs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hole6</td>
<td>42</td>
<td>133</td>
<td>294</td>
<td>600K</td>
<td>0.06</td>
<td>?</td>
<td>1120</td>
</tr>
<tr>
<td>hole7</td>
<td>56</td>
<td>204</td>
<td>448</td>
<td>7,200K</td>
<td>0.72</td>
<td>4.56</td>
<td>6.33</td>
</tr>
<tr>
<td>hole8</td>
<td>72</td>
<td>297</td>
<td>648</td>
<td>95,000K</td>
<td>9.50</td>
<td>54.98</td>
<td>5.79</td>
</tr>
<tr>
<td>hole9</td>
<td>90</td>
<td>415</td>
<td>900</td>
<td>1.4 G(est)</td>
<td>70.00</td>
<td>625.00</td>
<td>8.93</td>
</tr>
</tbody>
</table>

Notice that this does not include to time to compile, place and route and program the FPGA. While that time can make the speedup less than zero for small designs, as the # of variables gets larger, the hardware time (and size) doesn’t grow exponentially like time to complete does so speedup increases rapidly with size (see reference papers).
For larger designs in which times to solve problem by software are measured in many hours or days, hardware solving by this means may become feasible. i.e. If an hour of programming an FPGA saved you days of computer time

Things I would like to try in the future

- Add more complex logic to allow for faster solution (implication logic, etc)
- Find ways to minimize compilation and programming time. Leonardo is not the way to go, custom software would be better.
Sources

Jacob Boles
Class Project in
Fall 99