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 = $C_1 \bullet C_2 \bullet \ldots \bullet C_m$ 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

The Satisfiability Problem (cont.)

- 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.

Block Diagram of typical Machine

- 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.

Block Diagram of typical Machine

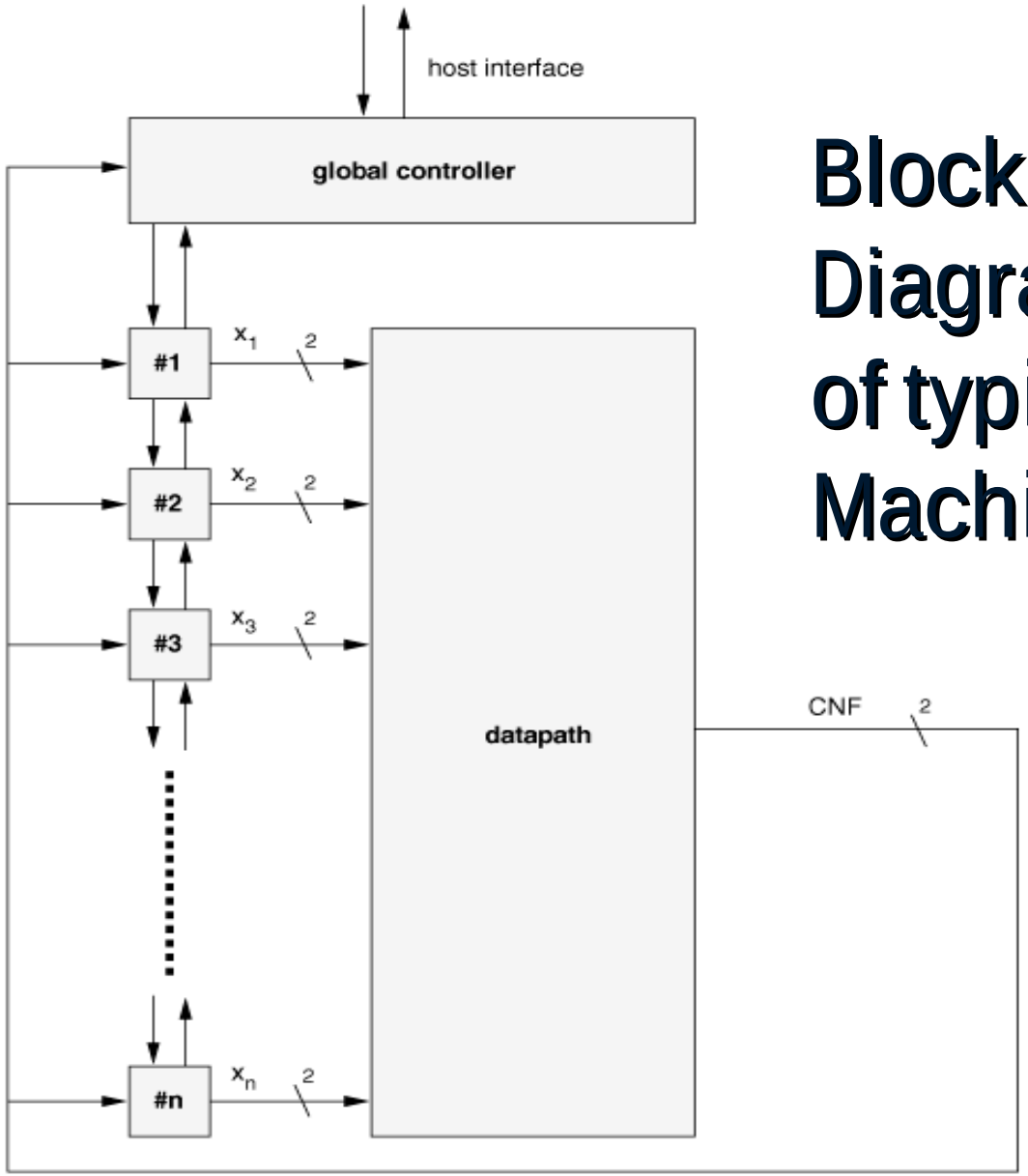
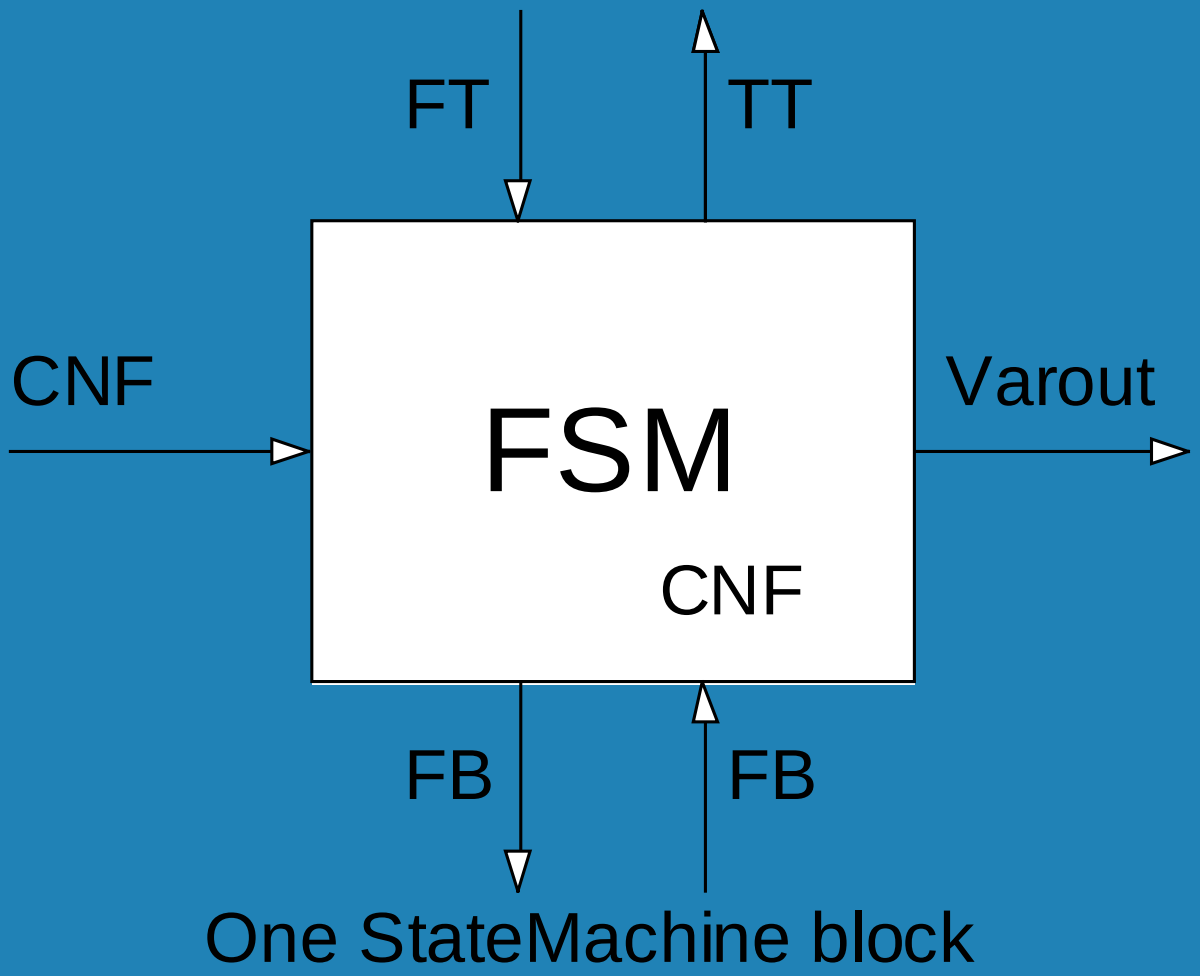


Fig. 1. Block diagram for the basic architecture (CE), consisting of an array of FSMs (#1 ... #n), a datapath, and a global controller. The variables x_i and the CNF are modeled in 3-valued logic.

State Machine for each variable



State Machine for each variable

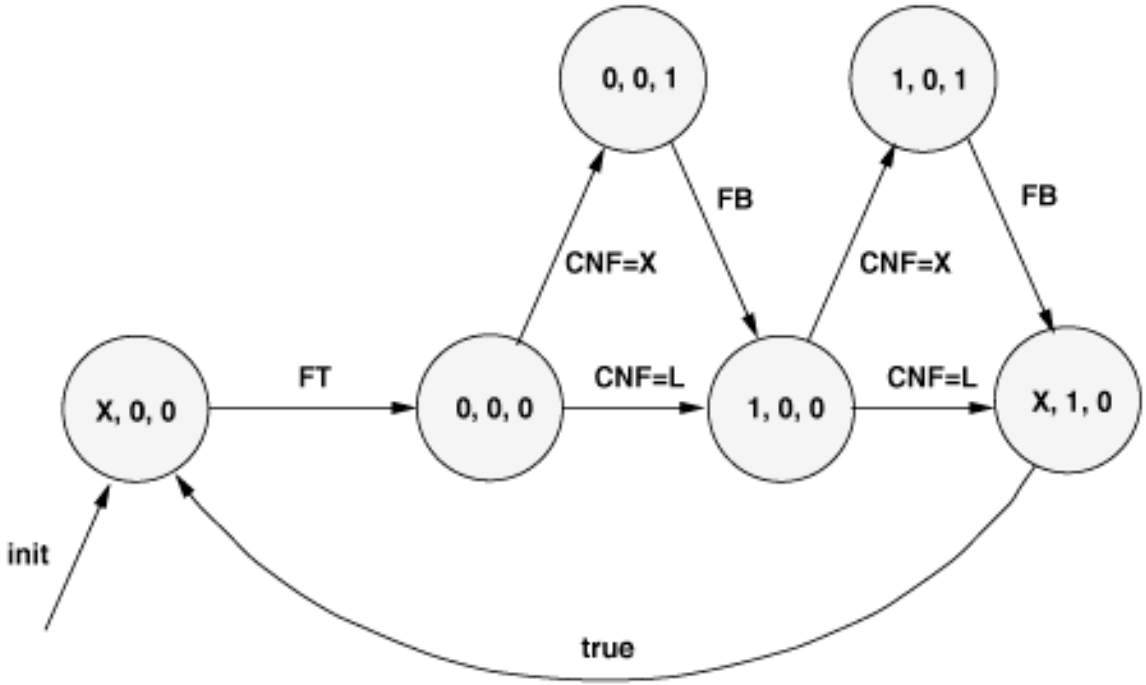
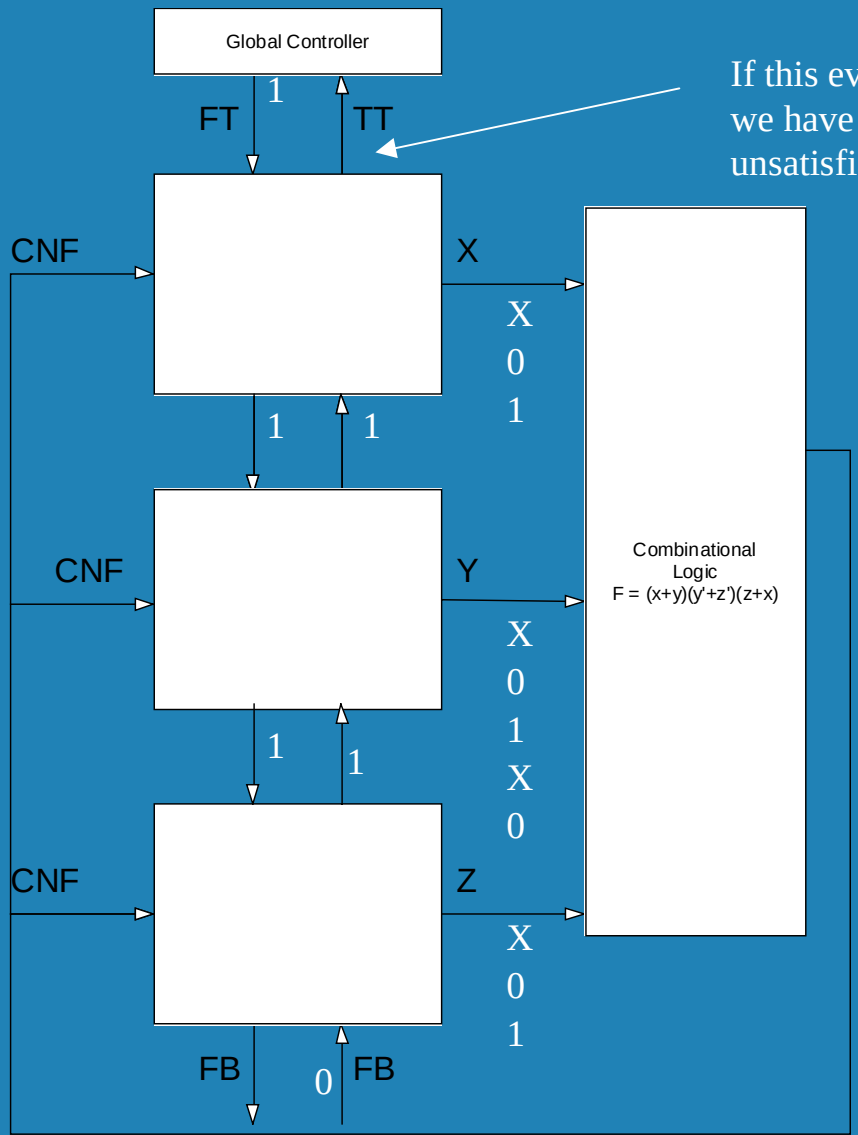


Fig. 2. State diagram for an FSM of the architecture CE. The inputs are FT (from top) and FB (from bottom) that activate the FSM, and the 3-valued CNF. The output signals displayed inside the states are the variable value, and the signals TT (to top) and TB (to bottom) that activate the previous and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

Simple Example

CNF = (x+y)(y'+z')(z+x)



F=(X+X)(X+X)(X+X) = X

F=(0+X)(X+X)(X+0) = X

F=(0+0)(1+X)(X+0) = 0

F=(0+1)(0+X)(X+0) = X

F=(0+1)(0+1)(0+0) = 0

F=(0+1)(0+0)(1+0) = 0

F=(1+1)(0+0)(1+1) = 0

F=(1+0)(1+0)(1+1) = 1

- 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!

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

```
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;     -- asynronous 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          : IN TERNARY;        -- signal out
    Vout         : BUFFER TERNARY
    );
END satis_control;

ARCHITECTURE structural OF satis_control IS

 -- Define all of 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;
  ELSIF ( 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 it's associated outputs.
  CASE present_state IS
   WHEN inactive =>

        Vout <= 'N'; to_top <= '0'; to_bot <= '0';
        IF ( from_top = '1' ) THEN
          next_state <= active_low;
        ELSE
          next_state <= inactive;
        END IF;

   WHEN active_low =>

        Vout <= '1'; to_top <= '0'; to_bot <= '0'; -- Vout <= 00
        IF( FIN = 'N' ) THEN
         -- no solution active next FSM
          next_state <= activate_bot_low;
        ELSIF( FIN = '0' ) THEN
         -- try 1 now
          next_state <= active_high;
        ELSE
          next_state <= inactive;
        END IF;

   WHEN active_high =>

         Vout <= '0'; to_top <= '0'; to_bot <= '0'; -- Vout <= 01
         IF( FIN = 'N' ) 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 <= '1'; to_top <= '0'; to_bot <= '1'; -- vout<= '00'
        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 <= '1'; -- vout <= 01
        IF( from_bot = '1' ) THEN
          next_state <= tried_both;
        ELSE
          next_state <= activate_bot_high;
        END IF;

   WHEN tried_both  =>

        Vout <= 'N'; to_top <= '1'; to_bot <= '0';
        next_state <= inactive;

  end case;
end process;
end structural;
```

VHDL for 3 valued logic

```
PACKAGE BODY ternary IS

  CONSTANT NOT_TABLE: TERNARY_1D :=
-  ('0', '1', 'N')
  ('1', '0', 'N');

  CONSTANT OR_TABLE: TERNARY_2D := (
-  ('0', '1', 'X')
  ('0', '1', 'N'),     -| 0 |
  ('1', '1', 'N'),     -| 1 |
  ('N', '1', 'N'));    -| X |

  CONSTANT AND_TABLE: TERNARY_2D := (
-  ('0', '1', 'X')
  ('0', '0', '0'),     -| 0 |
  ('0', '1', 'N'),     -| 1 |
  ('0', 'N', 'N'));    -| X |

 FUNCTION "NOT" (ARG: TERNARY) RETURN ternary IS
 BEGIN
   RETURN( NOT_TABLE(ARG) );
 END;

 FUNCTION "+" (L_ARG, R_ARG: TERNARY) RETURN ternary IS
 BEGIN
   RETURN( OR_TABLE(L_ARG, R_ARG) );
 END;

 FUNCTION "*" (L_ARG, R_ARG: TERNARY) RETURN ternary IS
 BEGIN
   RETURN( AND_TABLE(L_ARG, R_ARG) );
 END;

END ternary;
```

- Overloads operators so “+” “*” and NOT can be used for TERNARY signals.

C code to generate VHDL for logic

```
c FILE: aim-50-1_6-no-2.cnf
c
c SOURCE: Kazuo Iwama, Eiji Miyano (miyano@cscu.kyushu-u.ac.jp),
c         and Yuichi Asahiro
c
c DESCRIPTION: Artifical instances from generator by source.  Generators
c              and more information in sat/contributed/iwama.
c
c NOTE: Not Satisfiable
c
p cnf 50 80
5 17 37 0
24 28 37 0
24 -28 40 0
4 -28 -40 0
4 -24 29 0
13 -24 -29 0
-13 -24 -29 0
-4 10 -17 0
-4 -10 -17 0
26 33 -37 0
5 -26 34 0
33 -34 48 0
33 -37 -48 0
5 -33 -37 0
...
```

CNF Format file

- 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

```
int main(int argc, char* argv[])
{
    FILE    *cnf_file;
    FILE    *vhdl_file;
    char    s_buffer[256];
    char    s_numvar[20];
    int     n_numvar;
    char    s_numclause[20];
    int     n_numclause;
    bool    cnf_format = false;
    bool    sat_format = false;
    char    tmpstr[256];

    memset( s_numvar, '\0', sizeof( s_numvar ) );
    memset( s_numclause, '\0', sizeof( s_numclause ) );

    if( argc != 3 )
    {
        printf( "Command line is vhdl_cnf <filename in> <filename out>\n\n" );
        return(0);
    }

    if( (cnf_file = fopen( argv[1], "r" ) ) == NULL )
    {
        printf( "Error opening cnf file\n\n" );
        return(0);
    }

    // Copy our template vhdl file to the file specified
    if( (_access( argv[2], 0 )) != -1 )
    {
        printf( "File %s is already there, removing...\n", argv[2] );
        remove( argv[2] );
    }
    sprintf( tmpstr, "copy system.vhd %s", argv[2] );
    system( tmpstr );

    if( (vhdl_file = fopen( argv[2], "a" ) ) == NULL )
    {
        printf( "Error opening vhdl file\n\n" );
        return(0);
    }

    if( fgets( s_buffer, 256, cnf_file ) == NULL )
    {
        printf( "File error\n\n" );
        return(0);
    }
```

- 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

```
for(int i=0; s_buffer[0] == 'c' && i != 200;i++ )
{
    if( i == 200 )
    {
        // We didn't find the end of the comments
        printf( "File is nothing but comments\n\n" );
        return(0);
    }
    if( fgets( s_buffer, 256, cnf_file ) == NULL )
    {
        printf( "File error\n\n" );
        return(0);
    }
}

// We are past the comments
if( s_buffer[0] != 'p' )
{
    // No preamble line!
    printf( "Found no preamble line\n" );
    return(0);
}

if( tolower(s_buffer[2]) == 'c' && tolower(s_buffer[3]) == 'n'
    && tolower(s_buffer[4]) == 'f' )
{
    cnf_format = true;
}
else if( tolower(s_buffer[2]) == 's' && tolower(s_buffer[3]) == 'a'
    && tolower(s_buffer[4]) == 't' )
{
    sat_format = true;
    printf( "CNF format is supported only\n\n" );
    return(0);
}
else
{
    printf( "Error improper format defined in preamble statement\n\n" );
    return(0);
}
```

- 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

```
for( i=6;i<26;i++ )
{
    if( isdigit( s_buffer[i] ) )
    {
        s_numvar[i-6] = s_buffer[i];
    }
    else if( s_buffer[i] = ' ' )
    {
        break;
    }
    else
    {
        printf( "Invalid preamble\n\n" );
        return(0);
    }
}

for( int j=i+1;j<25;j++ )
{
    if( isdigit( s_buffer[j] ) )
    {
        s_numclause[j-i-1] = s_buffer[j];
    }
    else if( s_buffer[j] = ' ' )
    {
        break;
    }
    else
    {
        printf( "Invalid preamble\n\n" );
        return(0);
    }
}

n_numvar = atoi( s_numvar );
n_numclause = atoi( s_numclause );

printf( "Number variable: %d Number clauses: %d", n_numvar, n_numclause );

// first right out the signals needed to vhdl file
fprintf( vhdl_file, "SIGNAL from_top :  STD_LOGIC_VECTOR(%d DOWNTO 0);\n", n_numvar );
fprintf( vhdl_file, "SIGNAL to_top      :  STD_LOGIC_VECTOR(%d DOWNTO 0);\n", n_numvar );
fprintf( vhdl_file, "SIGNAL var_out     :  TERNARY_ARRAY(%d DOWNTO 1);\n", n_numvar);
fprintf( vhdl_file, "SIGNAL clause    :  TERNARY_ARRAY(%d DOWNTO 1);\n", n_numvar);

fprintf( vhdl_file, "BEGIN\nfrom_top(0) <= start;\nto_top(%d) <= '0';\n\n", n_numvar);
fprintf( vhdl_file, "\nFout <= " );
```

- 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

```
for( j=0;j<n_numclause;j++ )
{
    if( fgets( s_buffer, 256, cnf_file ) == NULL )
    {
        printf( "File error\n" );
        return(0);
    }
    if( s_buffer[0] == 'c' )
        continue;

    fprintf( vhdl_file, "(" );

    for( i=0;i<256;i++ )
    {
        if( s_buffer[i] == '-' )
        {
            fprintf( vhdl_file, "NOT ");
            i++;
        }

        fprintf( vhdl_file, "var_out(" );
        while( isdigit(s_buffer[i]) )
        {
            fprintf( vhdl_file, "%c", s_buffer[i] );
            i++;
        }

        if( s_buffer[i+1] == '0' )
        {
            break;
        }
        fprintf( vhdl_file, ") + " );
    }
    fprintf( vhdl_file, ")" );

    if( j != n_numclause-1 )
    {
        fprintf( vhdl_file, ") * " );
    }
}
fprintf( vhdl_file, ");\n" );

// output generate statement for control blocks
fprintf( vhdl_file, "\nCONTROL:\n" );
fprintf( vhdl_file, "FOR I IN %d DOWNTO 0 GENERATE\n", n_numvar-1 );
fprintf( vhdl_file, "BEGIN\n" );
fprintf( vhdl_file,
        "CX: matis_control PORT MAP( clk, reset_n, from_top(I), to_top(I+1),"
        "to_top(I), from_top(I+1), Fout, var_out(I+1) );\n" );
fprintf( vhdl_file, "END GENERATE CONTROL;\n" );
```

- 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

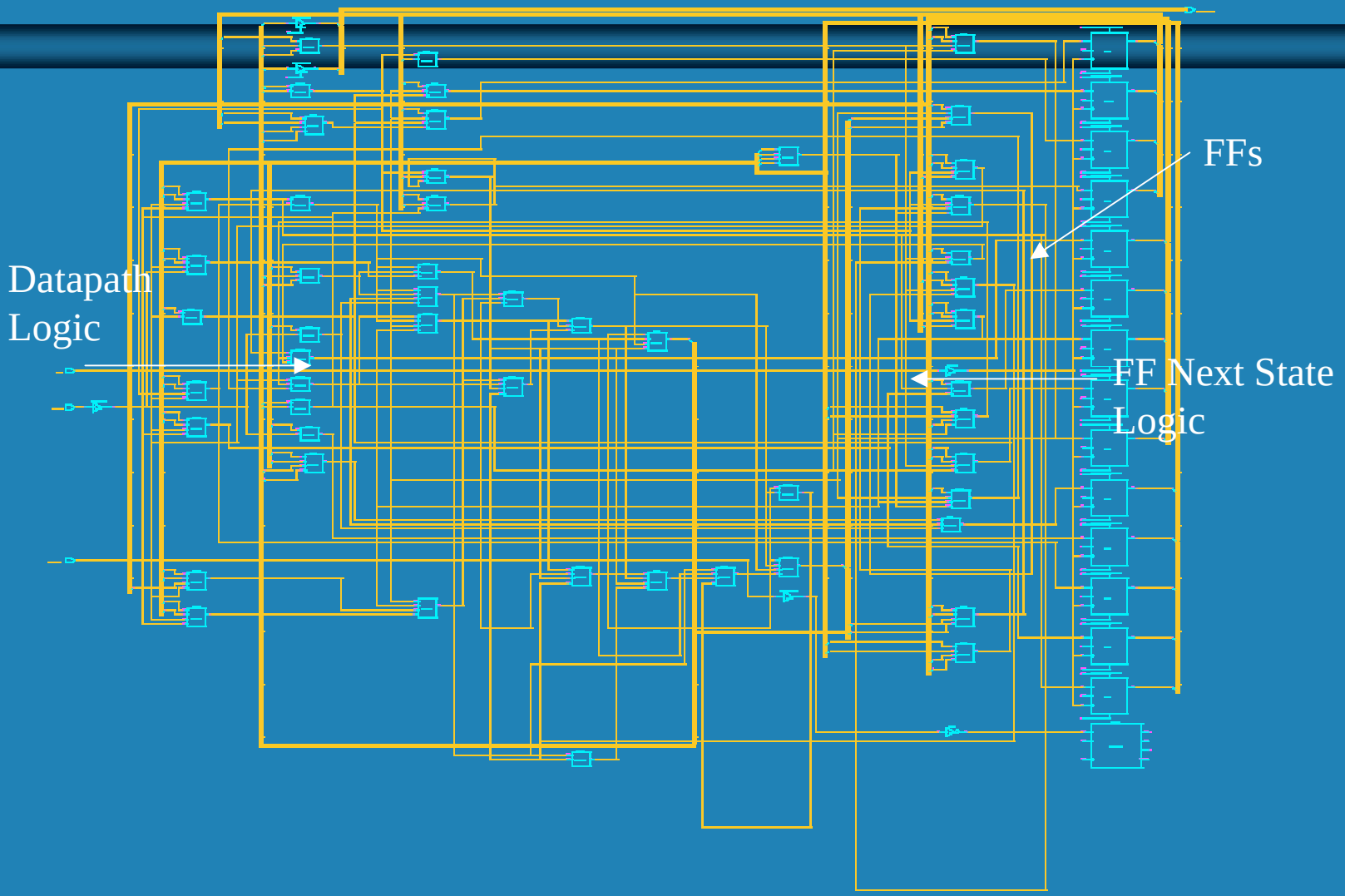
Example CNF File & output VHDL

```
c A simple example F =
      (x+y)(y'+z')(z+x)
p cnf 3 3
1 1 0
-2 -3 0
3 1 0
```

The above CNF file generates the VHDL code to the right

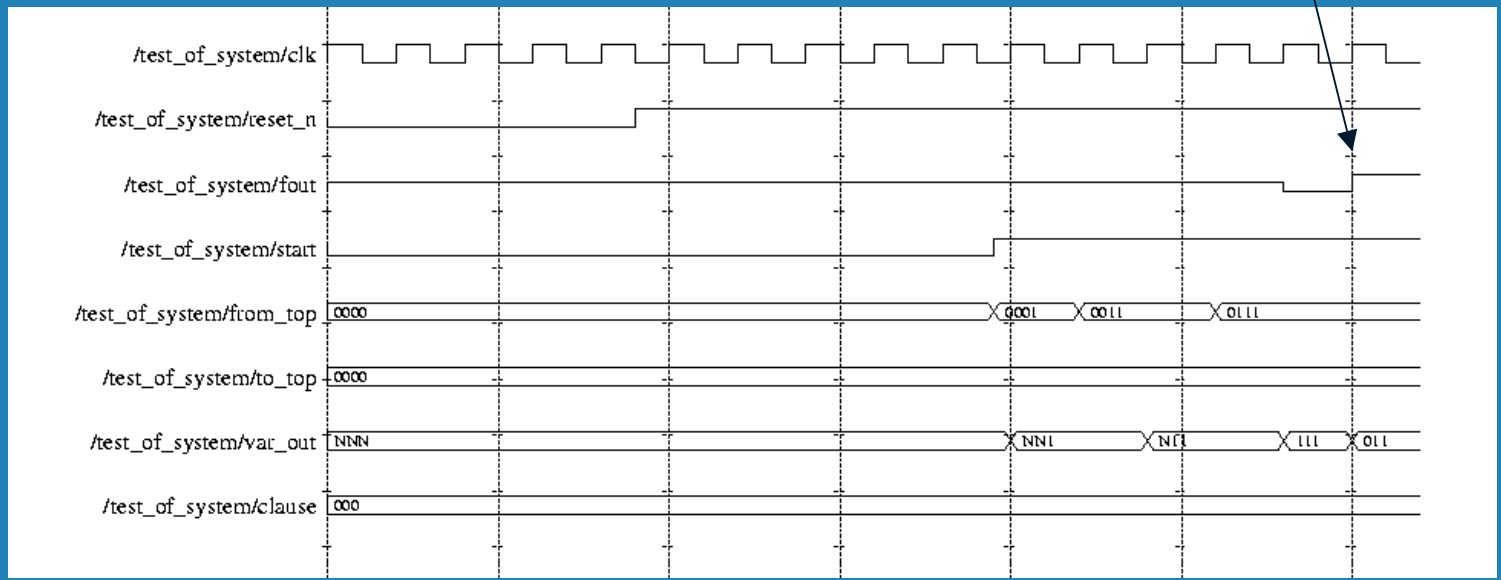
```
-- Jacob Boles
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE work.ternary.ALL;

ENTITY test_of_system IS
 PORT (
    clk       : IN    STD_LOGIC;    -- input clock
    reset_n   : IN    STD_LOGIC;    -- reset async active low
    Fout      : BUFFER TERNARY;
    start     : IN STD_LOGIC
    );
 END test_of_system;

ARCHITECTURE rtl OF test_of_system IS

 COMPONENT satis_control
 PORT (
     clk       : IN  STD_LOGIC;      -- input clock
     reset_n : IN  STD_LOGIC;      -- asynronous reset
     from_top    : IN  STD_LOGIC;      -- signal from higher FSM
     from_bot    : IN  STD_LOGIC;      -- signal from lower FSM
     to_top          : BUFFER STD_ULOGIC;  -- signal to higher FSM
     to_bot : BUFFER STD_LOGIC;    -- signal to lower FSM
     Fin         : IN TERNARY; -- signal to write to ram
     Vout        : BUFFER TERNARY
     );
 END COMPONENT;

SIGNAL from_top :  STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL to_top   :  STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL var_out  :  TERNARY_ARRAY(3 DOWNTO 1);
SIGNAL clause   :  TERNARY_ARRAY(3 DOWNTO 1);
BEGIN
from_top(0) <= start;
to_top(3) <= '0';


Fout <= (var_out(1) + var_out(2)) * (NOT var_out(2) + NOT var_out(3)) * (var_out(3) + var_out(1));

CONTROL:
FOR I IN 2 DOWNTO 0 GENERATE
BEGIN
CX: satis_control PORT MAP( clk, reset_n, from_top(I), to_top(I+1), to_top(I), from_top(I+1), Fout, var_out(I+1) );
END GENERATE CONTROL;

END RTL;
```

Schematic after synthesis



Timing Diagram for simple problem

Fout =1, solution found



- It solves the problem 10 cycles after it is started

A more complex example: hole6 of the DIMACS benchmark suite

c File: hole6.cnf
c
c SOURCE: John Hooker (jh38+@andrew.cmu.edu)
c
c DESCRIPTION: Pigeon hole problem of placing n (for file holen) pigeons
c in n+1 holes without placing 2 pigeons in the same hole
c
c NOTE: Part of the collection at the Forschungsinstitut fuer
c anwendungsorientierte Wissensverarbeitung in Ulm Germany.
c
c NOTE: Not satisfiable
c
p cnf 42 133
-1 -7 0
-1 -13 0
-1 -19 0
-1 -25 0

42 variables &
133 clauses!

```
-- eight bit BCD updown counter
-- Jacob Bides
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE work.ternary.ALL;

ENTITY test_of_system IS
 PORT (
    clk          : IN    STD_LOGIC;    -- input clock
    reset_n      : IN    STD_LOGIC;    -- reset async active low
    Fout         : BUFFER TERNARY;
    start        : IN  STD_LOGIC
    );
 END test_of_system;

 ARCHITECTURE rtl OF test_of_system IS

 COMPONENT satis_control
  PORT (
     clk          : IN  STD_LOGIC;      -- input clock
     reset_n : IN  STD_LOGIC;      -- asynnonous 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          : IN TERNARY; -- signal to write to ram
     Vout         : BUFFER TERNARY
     );
 END COMPONENT;

SIGNAL from_top :  STD_LOGIC_VECTOR(42 DOWNTO 0);
SIGNAL to_top   :  STD_LOGIC_VECTOR(42 DOWNTO 0);
SIGNAL var_out  :  TERNARY_ARRAY(42 DOWNTO 1);
SIGNAL clause   :  TERNARY_ARRAY(42 DOWNTO 1);
BEGIN
from_top(0) <= start;
to_top(42) <= '0';

Fout <= (NOT var_out(1) + NOT var_out(7)) * (NOT var_out(1) + NOT var_out(13)) * (NOT var_out(1) + NOT var_out(19)) * (NOT var_out(1) + NOT var_out(25))* (NOT var_out(1) + NOT var_out(31)) * (NOT var_out(1) + NOT
        var_out(37)) * (NOT var_out(7) + NOT var_out(13)) * (NOT var_out(7) + NOT var_out(19)) * (NOT var_out(7) + NOT var_out(25)) * (NOT var_out(7) + NOT var_out(31)) * (NOT var_out(7) + NOT
        var_out(37)) * (NOT var_out(13) + NOT var_out(19)) * (NOT var_out(13) + NOT var_out(25)) * (NOT var_out(13) + NOT var_out(31)) * (NOT var_out(13) + NOT var_out(37)) * (NOT var_out(19) + NOT
        var_out(25)) * (NOT var_out(19) + NOT var_out(31)) * (NOT var_out(19) + NOT var_out(37)) * (NOT var_out(25) + NOT var_out(31)) * (NOT var_out(25) + NOT var_out(37)) * (NOT var_out(31) + NOT
        var_out(37)) * (NOT var_out(2) + NOT var_out(8) * (NOT var_out(2) + NOT var_out(14)) * (NOT var_out(2) + NOT var_out(20)) * (NOT var_out(2) + NOT var_out(26)) * (NOT var_out(2) + NOT var_out(32))
        * (NOT var_out(2) + NOT var_out(38)) * (NOT var_out(8) + NOT var_out(14)) * (NOT var_out(8) + NOT var_out(20)) * (NOT var_out(8) + NOT var_out(26)) * (NOT var_out(8) + NOT var_out(32)) * (NOT
        var_out(8) + NOT var_out(38)) * (NOT var_out(14) + NOT var_out(20)) * (NOT var_out(14) + NOT var_out(26)) * (NOT var_out(14) + NOT var_out(32)) * (NOT var_out(14) + NOT var_out(38)) * (NOT
        var_out(20) + NOT var_out(26)) * (NOT var_out(20) + NOT var_out(32)) * (NOT var_out(20) + NOT var_out(38)) * (NOT var_out(26) + NOT var_out(32)) * (NOT var_out(26) + NOT var_out(38)) * (NOT
        var_out(32) + NOT var_out(38)) * (NOT var_out(3) + NOT var_out(9)) * (NOT var_out(3) + NOT var_out(15)) * (NOT var_out(3) + NOT var_out(21)) * (NOT var_out(3) + NOT var_out(27)) * (NOT var_out(3)
        + NOT var_out(33) * (NOT var_out(3) + NOT var_out(39)) * (NOT var_out(9) + NOT var_out(15)) * (NOT var_out(9) + NOT var_out(21)) * (NOT var_out(9) + NOT var_out(27)) * (NOT var_out(9) + NOT
        var_out(33)) * (NOT var_out(9) + NOT var_out(39)) * (NOT var_out(15) + NOT var_out(21)) * (NOT var_out(15) + NOT var_out(27)) * (NOT var_out(15) + NOT var_out(33)) * (NOT var_out(15) + NOT
        var_out(39)) * (NOT var_out(21) + NOT var_out(27)) * (NOT var_out(21) + NOT var_out(33)) * (NOT var_out(21) + NOT var_out(39)) * (NOT var_out(27) + NOT var_out(33)) * (NOT var_out(27) + NOT
        var_out(39)) * (NOT var_out(33) + NOT var_out(39)) * (NOT var_out(4) + NOT var_out(10)) * (NOT var_out(4) + NOT var_out(16)) * (NOT var_out(4) + NOT var_out(22)) * (NOT var_out(4) + NOT
        var_out(28)) * (NOT var_out(4) + NOT var_out(34) * (NOT var_out(4) + NOT var_out(40)) * (NOT var_out(10) + NOT var_out(16)) * (NOT var_out(10) + NOT var_out(22)) * (NOT var_out(10) + NOT
        var_out(28)) * (NOT var_out(10) + NOT var_out(34) * (NOT var_out(10) + NOT var_out(40)) * (NOT var_out(16) + NOT var_out(22)) * (NOT var_out(16) + NOT var_out(28)) * (NOT var_out(16) + NOT
        var_out(34)) * (NOT var_out(16) + NOT var_out(40)) * (NOT var_out(22) + NOT var_out(28)) * (NOT var_out(22) + NOT var_out(34)) * (NOT var_out(22) + NOT var_out(40)) * (NOT var_out(28) + NOT
        var_out(34)) * (NOT var_out(28) + NOT var_out(40)) * (NOT var_out(34) + NOT var_out(40)) * (NOT var_out(5) + NOT var_out(11)) * (NOT var_out(5) + NOT var_out(17)) * (NOT var_out(5) + NOT
        var_out(23)) * (NOT var_out(5) + NOT var_out(29)) * (NOT var_out(5) + NOT var_out(35)) * (NOT var_out(5) + NOT var_out(41)) * (NOT var_out(11) + NOT var_out(17)) * (NOT var_out(11) + NOT
        var_out(23)) * (NOT var_out(11) + NOT var_out(29)) * (NOT var_out(11) + NOT var_out(35)) * (NOT var_out(11) + NOT var_out(41)) * (NOT var_out(17) + NOT var_out(23)) * (NOT var_out(17) + NOT
        var_out(29)) * (NOT var_out(17) + NOT var_out(35)) * (NOT var_out(17) + NOT var_out(41)) * (NOT var_out(23) + NOT var_out(29)) * (NOT var_out(23) + NOT var_out(35)) * (NOT var_out(23) + NOT
        var_out(41)) * (NOT var_out(29) + NOT var_out(35)) * (NOT var_out(29) + NOT var_out(41)) * (NOT var_out(35) + NOT var_out(41)) * (NOT var_out(6) + NOT var_out(12)) * (NOT var_out(6) + NOT
        var_out(18)) * (NOT var_out(6) + NOT var_out(24)) * (NOT var_out(6) + NOT var_out(30)) * (NOT var_out(6) + NOT var_out(36)) * (NOT var_out(6) + NOT var_out(42)) * (NOT var_out(12) + NOT
        var_out(18)) * (NOT var_out(12) + NOT var_out(24)) * (NOT var_out(12) + NOT var_out(30)) * (NOT var_out(12) + NOT var_out(36) * (NOT var_out(12) + NOT var_out(42)) * (NOT var_out(18) + NOT
        var_out(24)) * (NOT var_out(18) + NOT var_out(30)) * (NOT var_out(18) + NOT var_out(36)) * (NOT var_out(18) + NOT var_out(42)) * (NOT var_out(24) + NOT var_out(30)) * (NOT var_out(24) + NOT
        var_out(36)) * (NOT var_out(24) + NOT var_out(42)) * (NOT var_out(30) + NOT var_out(36)) * (NOT var_out(30) + NOT var_out(42)) * (NOT var_out(36) + NOT var_out(42)) * (var_out(6) + var_out(1) +
        var_out(4) + var_out(3) + var_out(2) + var_out(1))* (var_out(12) + var_out(11) + var_out(10) + var_out(9) + var_out(8) + var_out(7)) * (var_out(18) + var_out(17) + var_out(16) + var_out(15) + var_out(14) +
        var_out(13)) * (var_out(24) + var_out(23) + var_out(22) + var_out(21) + var_out(20) + var_out(19)) * (var_out(30) + var_out(29) + var_out(28) + var_out(27) + var_out(26) + var_out(25)) * (var_out(36) +
        var_out(35) + var_out(34) + var_out(33) + var_out(32) + var_out(31)) * (var_out(42) + var_out(41) + var_out(40) + var_out(39) + var_out(38) + var_out(37));


CONTROL:
FOR I IN 41 DOWNTO 0 GENERATE
BEGIN
CX: satis_control PORT MAP( clk, reset_n, from_top(I), to_top(I-1), to_top(I), from_top(I-1), Fout, var_out(I+1) );
END GENERATE CONTROL;

END RTL;
```

Not meant to be readable
Notice though how long the datapath logic is compared to previous example

Schematic is 4 pages long!

Mapping report for hole6

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
****

Resource	Used	Avail	Utilization
IOs	5	129	3.88%
FG Function Generators	1120	1152	97.22%
H Function Generators	134	576	23.26%
CLB Flip Flops	251	1152	21.79%

Clock	: Frequency
clk	: 2.0 MHz

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

Performance Comparison of my Machine to Software

- 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

Benchmark	#inputs	#clauses	#literals	#Clocks	time @10MHz	GRASP time on 300Mhz Pent	SU	Cost(FGs)
hole6	42	133	294	600K	0.06	?	?	1120
hole7	56	204	448	7,200K	0.72	4.56	6.33	1389
hole8	72	297	648	95,000K	9.50	54.98	5.79	2139
hole9	90	415	900	1.4 G(est)	70.00	625.00	8.93	Didn't complete

- 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).

Conclusion/Future Work

- 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