CS 254 Project

FPGA based SAT-Solver

2015-16

April 8, 2016

Group Members

Roll Number	Member
140050003	Harshal Mahajan
140050005	Rupanshu Ganvir
140050081	Sumith
140050086	Shubham Goel

Contents

1 Introduction

1.1 Problem Statement

Our aim is to design an FPGA based SAT Solver. It takes in a CNF formula and reports whether the formula is satisfiable and if so returns a satsifying assignment.

1.2 Aim

Main aim will be to build a SAT solver in VHDL and program it onto a FPGA board. The following are the specifics:

- Complete Algorithm: We aim to have a complete algorithm. We will build upon a basic DPLL algorithm and ensure completeness
- Optimization: Variable selection heuristics(e.g. VSIDS) which helps in choosing the decision variable(explained below).

1.3 Approach

We have used a form of the DPLL algorithm. A complete algorithm guarantees either to find a solution on termination or to prove that there is no solution. We ensure that it's a complete algorithm. Here, the search process is organized by implicitly traversing the space of all possible assignments of values to variables(psuedo code below). The recursive algorithm for DPLL is given below:

```
    procedure DP
    input: formula f in CNF, assignment M initially empty
    output: true or false, satisfying assignment M if true

    propogate()

        if(f contains empty clause)
            return false
        if(f contains no clause)
            return true with M

        v = DecideNextVariable()

        DP(f with p=True, M with p=True)
        DP(f with p=False, M with p=False)

        return false
```

2 High Level Architecture

The major problem in adopting the above code is it is recursive and thus our first challenge was to design an algorithm along the similar lines. The iterative version will have exponential space complexity as well time complexity as the memory cannot be freed up after iteration

```
procedure DPLL-iterative
input: formula f in CNF, assignment M initially empty
output : true or false, satisfying assignment M if true
while (true)
    while (status = continue)
        status = reduction();
    end while;
    if (status = unknown)
        decideNextBranch ();
    else if (status = conflict)
    if (backtrack() = false)
        return unsat;
    else if (status = true)
        return sat, assignment;
    end if;
end while;
```

2.1 Overiew

2.1.1 Reduction

We are using two usually known methods for reduction. These methods have an advantage as they are constraints fulfilled by all the satisfying assignments and they are linear in time to detect.

- **FindUnitClause**: In this reduction technique, a clause containing exactly one unassigned literal is taken and a corresponding value is assigned - 1 if it is a positive literal, 0 if it is a negative literal.
- **FindPureLiteral**: If an atomic propsition occurs only as a postive(negative) literal throughout the CNF then it is assigned 1(0). Such an atomic proposition corresponds to a pure literal.

When no more reduction can be done, a decision is taken. A decision is basically assumption of assigning a value to an unassigned variable and then inferring from that assignment.

If we conclude with a satisfying assignment, we are successful else we have hit a conflict, we then have to backtrack and change the value assigned to the decision variable or change the variable itself if both values have been exhausted. This is evident from the recursion.

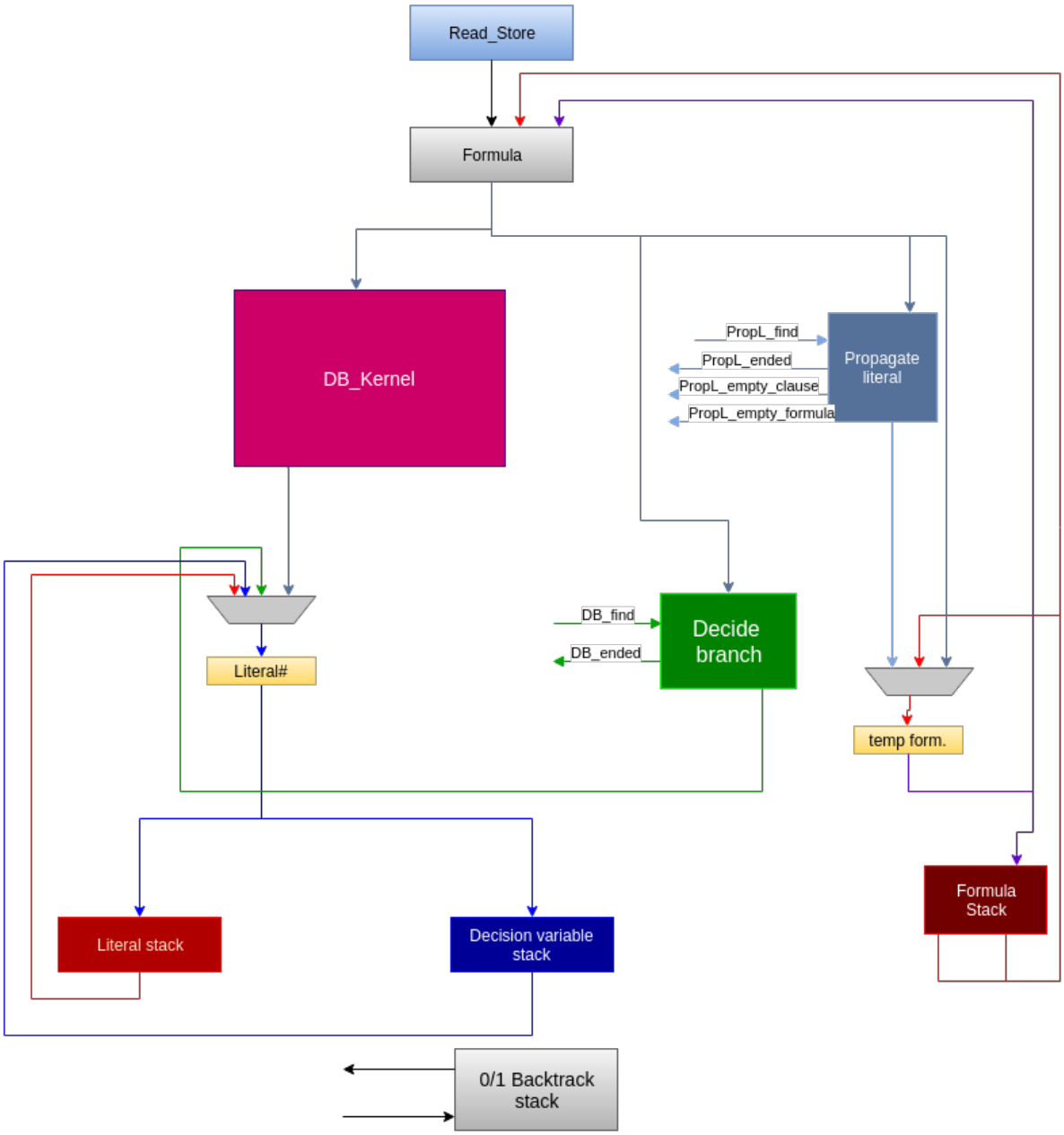
When a variable is assigned a value, all clauses satisfied can be removed. Also all literals that turn false after this assignment can be removed from their respective clauses. This may uncover more unit clauses and pure literals. Further reduction is applied. This process is called propagation.

2.1.2 Deciding the Branch and Backtracking

A conflict occurs when all the elements of a clause are falsified(removed), hence the clause is now empty. Whenever a conflict(empty clause) occurs we erase all actions till the most recent decision variable and invert it's value. If both values have been tried, then go back to previous decision variable.
The formula is SAT is when each clause is satisfied simultaneously i.e. removed from the formula, it means that the current variable assignment represents a solution.

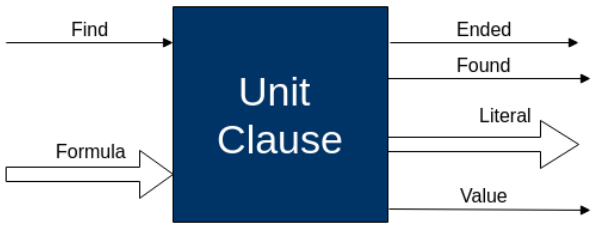
If all possible assignments of values to variables have been implicitly tested i.e. both values of the first decision variable were tried out without success, then the formula is UNSAT.

Descriptions and functionalities of blocks

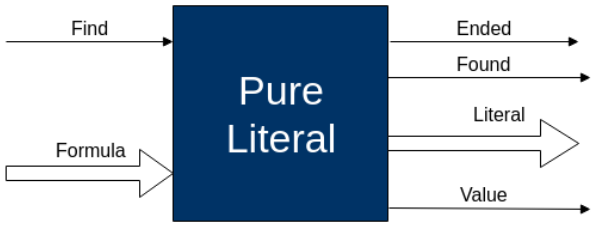


Following are the description of the individual blocks used:

- **Unit Clause**: Included in the DB_kernel. It takes in a $formula$ and is triggered to start operation when $find$ is high. A high *ended* signal signifies that the unit clause has completed its operation. Once the operation has *ended*, if $found$ signal is high then the input $formula$ indeed has a unit clause and so the *literal#* and *value* signals have the unit clause's information. If $found$ is low then the formula has no unit clause.



- **Pure Literal**: Included in the DB_Kernel. The input/output is same as that of the Unit Clause block. The output is same as above except it finds a pure literal.



- **Propagate Literal**:(Included in DB kernel as well) Takes in a $formula$ and is triggered to operate by $find$. It takes in a *variable#*, its *value* and reduces the formula with this assignment. Satisfied clauses are removed from the formula and falsified literals are removed from the clauses containing them, the resulting output $formula$ is passed. *check* is 0 by default, 1 if the resulting formula has a empty clause and 2(10) if no clauses are left in the formula.

- **Decide Branch**: Inputs are $formula$ and $find$. $literal\#$ and $value$ return the variable and value decided for that variable. *ended* signifies if the block has completed operation. This block can have a lot of improvisations. Various heuristics can be implemented for selecting the variable and it's value. These will be implemented in the future.

Apart from this some major memory elements needed(in addition to a few temporary variables) are the following:

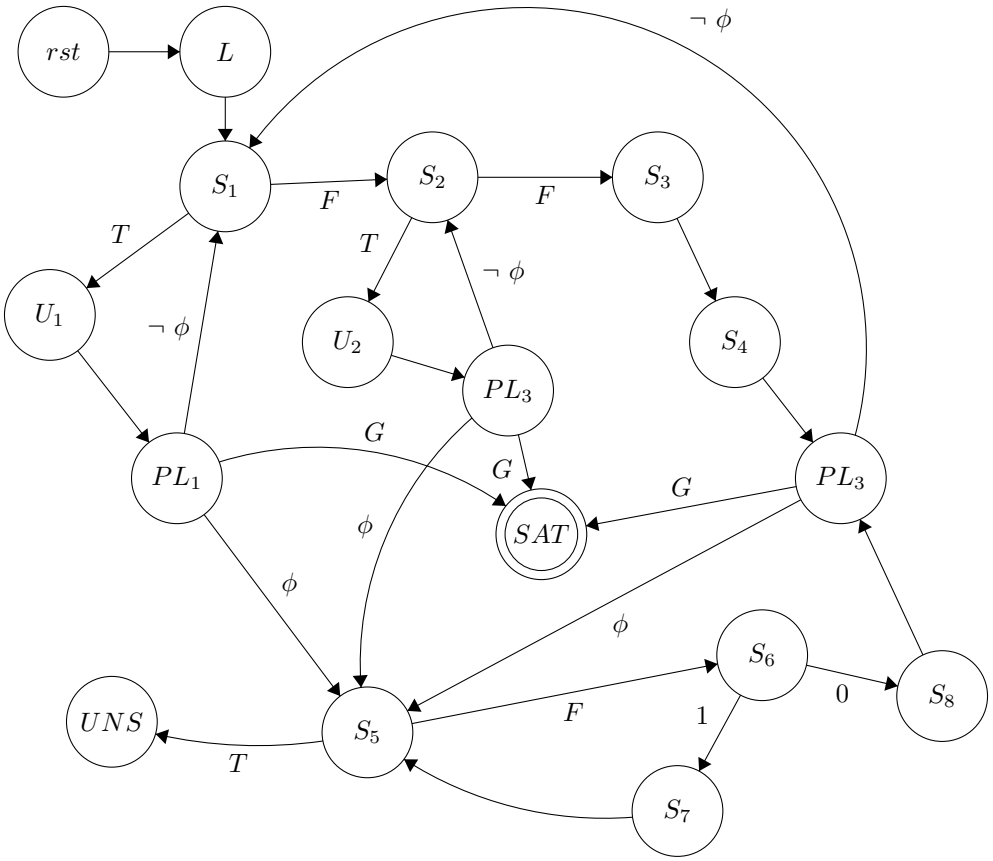
1. **Literal Stack**:All the variables that have been assigned until now. These variables may have been either assigned by inference or by decision.

2. **Decision variable stack**:All the variables that have been assigned by decision. This stack will help in easy backtracking

3. **0/1 Backtrack Stack**: This stack has either a 0 or a 1 corresponding to every variable in the decision variable stack. On a making a decision for a variable, 0 is pushed into the stack. If a conflict occurs, while backtracking we pop and insert 1 to the stack to signify one inversion of this decision variable has already been done. If it is 1, then both values have been tried, so we pop the entry and go back to previous decision variable.

4. **Formula stack**: After every decision, we propagate and hence simplify the formula. If the decision results in a conflict, the previous formula is needed. Hence prior to every decision we store the formula. This can hence be recovered after a conflict. We may try to optimize memory here by storing partial formulae.

2.2 State Transition Diagram (Overview)



Note: A blank transition here implies that the state work has finished and the 'ended' has been returned to the controlled.

2.2.1 Explanation

1. **rst**: Reset State
   The standard reset in a VHDL circuit. In this state reset all the variables of the circuit.

2. **L**: Load
   This is the **Load** variable as specified in the problem statement. We taken input in this state.

3. $\boldsymbol{S_1}$: Check Presence of Unit Clauses
   In this state, FindUnitClause is processing the formula. If there exists a unit clause, then it goes to U1, else it goes to S2.

4. $\boldsymbol{S_2}$: Check Pure Literal
   In this state, FindPureLiteral is processing the formula. If there exists a pure literal, then it goes to U2, else it goes to S3.

5. $\boldsymbol{S_3}$: Select the decision variable
   After all the reduction possible, we now take a decision in state S3. A variable and it's value is selected.

6. $\boldsymbol{S_4}$: Populate the Formula Stack
   In this state, we store the current formula in the formula stack before the propagating the decision result on the formula.

7. $\boldsymbol{S_5}$: Check Decision Variable is Empty
   This state checks the size of the Decision variable stack. If not empty, we go to S6. If this is empty, the formula is unsatisfiable. We return UNSAT and go to state UNS.

8. $\boldsymbol{S_6}$: Pop the 0/1 Bactrack Stack
   Pop the 0/1 Backtrack 0/1 Backtrack Stack. If the top of the stack was 0 go to state S8 else go to state S7

9. $\boldsymbol{S_7}$: Pop the Decision Variable Stack.
   Both the values of the decision variable have been tried. This decision variable lead to a conflict. Go back to the previous decision variable by popping the stack. Go to state S5.

10. $\boldsymbol{S_8}$: Negate the Decision Variable.
    Since only value of the last decision variable was tried, we invert the decision variable. We pop the 0/1 Backtrack stack and insert 1 to the 0/1 Backtrack stack to denote that this is the second try. We propagate the formula for the new assignment and go to PL3

11. $\boldsymbol{PL_1}$: Propagate Literal
    Here the previous assignment of the variable is used to reduce the formula. If the all the clauses are satisfied we go to SAT state and return the complete variable assignment. If any clause turned out false, then our assumption is wrong, we backtrack i.e. go to state 5. If the propagation is complete we try to find more unit clauses as a result of this propagation i.e. go to state S1.

12. $\boldsymbol{PL_2}$: Propagate Literal
    Same offering as previous PLs

13. $\boldsymbol{PL_3}$: Propagate Literal
    Same offering as previous PLs

14. $\boldsymbol{U_1}$: Update the literal stack

15. $\boldsymbol{U_2}$: Update the literal stack

16. **G** : Check if there are no clauses left
    G transitions from PLs happening when all clauses are removed as a result of propagation. This implies are clauses are true and hence statisfiable.

17. **UNS** : The formula is unsatisifiable
    The formula is unsatisfiable, return UNSAT.

18. **SAT** : The formula is satisifiable
    The formula is satisfiable, return SAT and the satisfying assignment

3 Assumptions and Constraints

3.1 Overall Input

The default values for the sizes of the arrays is the 64x1000. Thus it would result in a large space complexity and thus would slow down the code.

This can be changed in the *common.vhd* module in the constants where we have specified them.

4 Changes

4.1 Addressing Instructor Feedback

The feedback we received was "Interesting design, seems to require a lot of data transfer though between blocks". The immediate but not complete fix to this we thought was store in sparse representation instead of dense ,i.e. the formula $(-1+2).(3+4).(1+-3)$ where 1,2,3 and 4 are boolean variables

	1	2	3	4
1	-1	1	0	0
2	0	0	1	1
3	1	0	-1	0

This is the dense representation.

	1	2	3	4
1	-1	2	0	0
2	3	4	0	0
3	1	-3	0	0

We chose this because we can terminate the row iterations quickly as compared to dense, hence saving time.

But again due to lack of flexibility we had to declare (NUMBER_CLAUSES x NUMBER_LITERALS) arrays increasing the memory consumption and transfer. This is one drawback of our design.

5 Implementation

5.1 Files Submitted

1. Report
2. README
3. Testbenches
4. tussle_top.vhd (Controller)
5. Common.vhd (Package)
6. Unit_Clause.vhd
7. Pure_Literal.vhd
8. Propogate_Literal.vhd
9. DB_Kernel.vhd
10. decide_branch.vhd
11. Stack_bool.vhd
12. Stack_integer.vhd
13. Stack_formula.vhd
14. read_store.vhd

5.2 Style

The following stylings and coding aesthetics are implemented:

1. Intuitive nomenclature of variables
2. Comments for explanation of every varible and process
3. Port Mapping has format 'Component Name'_'Variable Name'
4. Appropriate indentation with Tab Size: 4
5. Signal in each module has name format 's'_'in/out variable name'

5.3 Data Structure

5.3.1 Package

Module Name: *Common.vhd* Types Defined:

- **lit**: Class for Literals
  - **lit_num**: *integer*; stores value from 0 to 64; indicates the number assigned to the literal
  - **val**: *STD_LOGIC*; stores whether positive or negative literal
- **clause**: Class for Clause
  - **lits**: *array of type lit*; stores the literals in the clause
  - **len**: *integer*; stores the number of literals in the clause
- **formula**: Class for Formula
  - **clauses**: *array of type clause*; stores the clauses in the formula
  - **len**: *integer*; stores the number of literals in the clause

5.3.2 Stacks

We are using stacks for 3 datatypes: *STD_LOGIC, INTEGER, FORMULA*. For this explanation purposes the core structure for all are same. Henceforth for this subsubsection, we would specify them as "$(type)"
Module Name: *Stack_$(type).vhd*
Input:

- **wr_en**: *STD_LOGIC*
- **pop**: *STD_LOGIC*
- **din**: *$(type)*

Output:

- **empty**: *STD_LOGIC*
- **full**: *STD_LOGIC*
- **dout**: *$(type)*
- **front**: *$(type)*

Function: Maintains a stack of $(type).
Implementation: It has the following datatypes:

- Data size: Maintaining the size of the array
- Array of Stack size defined in *common.vhd*

The pseudocode for the push and pop operations for the stacks:

```
/**PUSH*/
if (wr_en & !full)
    add to array(curr_size);
end if;
/**POP*/
if (pop & !empty)
    output array(curr_size);
    curr_size --;
end if;
```

5.4 Modules

We have tried to insert as much inline comments as possible. All modules have been commented. Please refer there to know the exact functioning of each entity. Every module has a reset and a clock pin. The rest of the inputs and outputs are mentioned below.

5.4.1 Input

Module Name: *read_store.vhd*
Input:

- **load**: *STD_LOGIC*
- **i**: *STD_LOGIC_VECTOR(64 bit)*

Output:

- **ended**: *STD_LOGIC*
- **formula_res**: *formula*

Function: Reads the input and stores it in a formula datatype

5.4.2 Deciding Branch Kernel

It contains 3 components: Pure Literal, Propogate Literal, and Unit Clause Module Name: *Stack_$(type).vhd*
Input:

- **find**: *STD_LOGIC*
- **pop**: *STD_LOGIC*
- **din**: *$(type)*

Output:

- **empty**: *STD_LOGIC*
- **full**: *STD_LOGIC*
- **dout**: *$(type)*
- **front**: *$(type)*

Function: Maintains a stack of $(type).

5.4.3 Decide Branch

Module Name: decide_branch.vhd Input:

- **find**: *STD_LOGIC*
- **in_formula**: *formula*

Output:

- **ended**: *STD_LOGIC*
- **out_lit**: *lit*

Function: Maintains a stack of $(type).

5.4.4 Controller (Top)

Module Name: *tussle_top.vhd* Input:

- **load**: *STD_LOGIC*
- **i**: *STD_LOGIC_VECTOR(64 bit)*

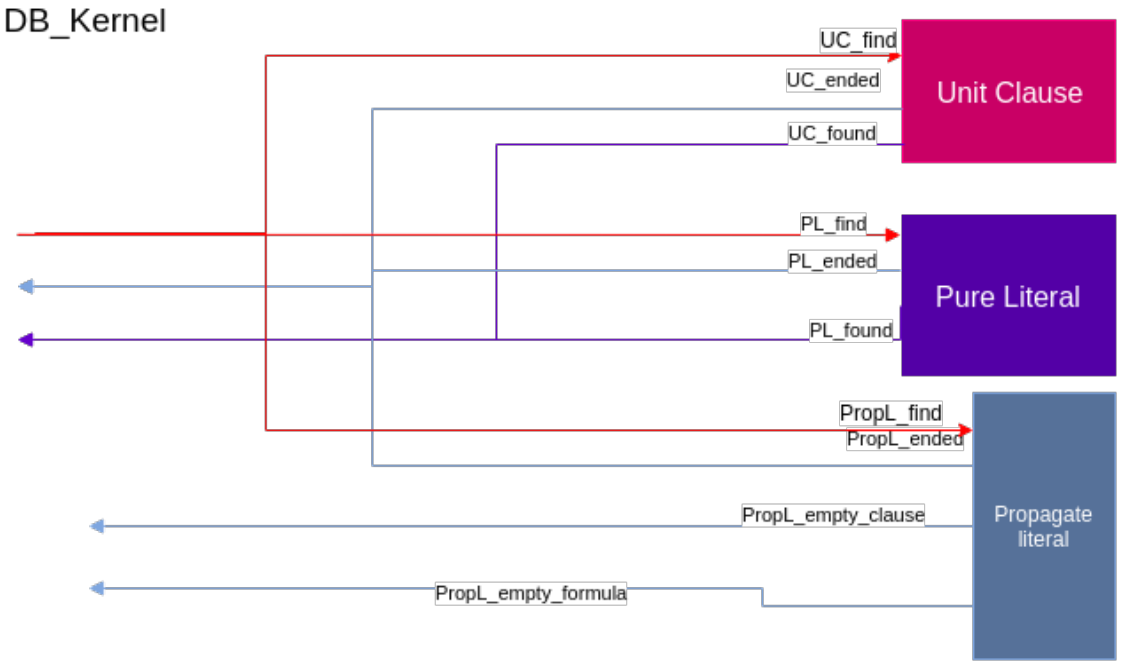
Output:

- **ended**: *STD_LOGIC*
- **sat**: *STD_LOGIC*
- **i**: *STD_LOGIC_VECTOR(64 bit)*

Function: Top Module of the project

5.5 Algorithm Implementation

Input is according the format the user has to input bit vectors of number of literals.The default value is 64. The input is stored in the form of packages with a 1000 clauses as the default value. We have to manually change the size of the clauses and variables to reduce the space complexity for various problems.



5.5.1 Decide Branch Kernel

The process starts with an empty variable assignment. Then unit clause rules, pure literal rules, and decisions(on unassigned variables) are applied sequentially. The propogate literal takes the lit and assigns itvalue accordingly and returns the formula. It also returns if a clause is empty or a formula is empty

```
while(true)
    if (find && idle)
        input in_formula;
        update states;
    else if (!idle)
        unit_clause();
        if (UC_found)
            propogate_literal();
            check_satisfiability;
        else
            pure_literal();
            if (!PL_found)
                propogate_literal();
                check_satisfiability;
```

```
            else
                return in_formula;
            end if;
        end if;
    end if;
end while;
```

5.5.2 Decide Branch

If after applying all the above rules we arrive at a confusion of assignment, we assign a value to the first clause with a literal. If we see a conflict, then we backtrack to change the value of the most recent variable.

```
while(true)
    if (find & idle)
        idle = false;
    else if (!idle)
        for all clauses:
            if clause_length != 0;
                select first lit;
            else
                iterate;
            end if;
        end for;
    end if;
end while;
```

5.5.3 Backtracking

The backtracking we implemented requires the use of 5 stacks. However the backtracking section is incorporated in the controller

5.6 Controller

The categories of the states for the controller are as follows:

5.6.1 Input

1. **Loading State and Input State**: When load is true this state is activated. It accepts input and stores it.

5.6.2 DB_Kernel

2. **Kernelize**: Supplies the formula to the DB_Kernel and starts Kernelizing state.

```
if(ended)
    Store formula;
    if (current SAT)
        return model;
    else if (current UNSAT)
        Backtrack by popping Decision variable stack;
    else
        decide_branch();
    end if;
end if;
```

5.6.3 BackTrack

3. **Backtrack**: Backtracks the temporary unsatisfiable formulae, if both the valueshave been exhausted if backtrack Stack has front as 1

```
if(backtrack stack empty)
    return UNSAT;
else if backtrack Stack has front 1
    pop from decision variable, backtrack, and formula stack
else
    Pop_Lit stack_State();
end if;
```

4. **Pop Literal Stack state**:

```
if (Lit_Stack_front and Decision Variable Stack front are Same)
    Pop_literal Stack;
    Store the front of the Decision Variable and Formula Stack;
    Pop Decision Variable Stack, Backtrack Stack, Formula Stack;
    Negate();
end if;
```

5. **Negate**: We negate before we run the propogate Literal and thus cover the backtracking branch by storing the value in the backtracking stack and then propogate.

6. **Propogate**: We update the input formulae and the literal. After propogating, we store the the temporary SAT state and check again

5.6.4 Decide Branch

7. **Decide Branch Calculations**: In this state we are calculating the pure literals, Unit Clauses and their formulae after substitution in the Formula.

5.7 Output States

8. **Return Model**: It completes the assignemnt from the literal stack. If literal stack is empty then we return SAT else it assigns all the values assigned in the literal stack

9. **SAT states**: They return the SAT states

6 Testing and Verification

We tested the code for few nominal variables that is less than 10, and for 20 clauses. The output terminated in less than 1000 clock cycles. We also tried for larger number of variables and the results were working for almost all the testcases. We have included the testcases as well as the test case generator "Syn_testbench.py".

7 Future Scope

As you progress, use the current model to get the formula back by substitution instead of storing various instances of the formula at different times. This seems the best solution we have but we couldn't rewrite the whole design in time. Store the diff of the formula instead of the whole formula itself at different stages. This will result in lesser memory consumption. Divide the formula into blocks and store diff of these blocks, this does better because at the later stages of the algorithm some blocks might not change at all and hence using lesser memory.We agree that our design is lacking in this areas and huge test cases might consume too much memory simulate. But all the test cases that have terminated have given output successfully.

8 Work Distribution

Shubham Goel

- Core structure(package Common)
- Stacks
- unit_clause
- propagate_literal
- DB_Kernel
- Top Pseudo Code
- Testing/Debugging

Sumith

- input parsing
- use common
- decide branch
- pure_literal
- Top level
- syntestbench.py
- Final packaging

Harshal

- Testing
- Documentation

Rupanshu

- Implemented variable selection heuristics(JSOS and DLIS)

References

[1] Robert Brummayer, Florian Lonsing and Armin Biere. *Automated Testing and Debugging of SAT and QBF Solvers* http://fmv.jku.at/papers/BrummayerLonsingBiere-SAT10.pdf

[2] Roberto Sebastiani *SAT: Propositional Satisfiability and Beyond* http://www.inf.ed.ac.uk/teaching/courses/propm/papers/main_lan1.pdf

[3] http://www.gecode.org/events/acp-summerschool-2011/slides/Gent/SATCP3.pdf

[4] Iouliia Skliarova and Antonio de Brito Ferrari *Reconfigurable Hardware SAT Solvers: A Survey of Systems* http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1336765&tag=1

[5] Niklas Een, Niklas Sorensson *An Extensible SAT-solver* http://minisat.se/downloads/MiniSat.pdf

[6] Kristopher Micinski *SAT: Propositional Satisfiability and Beyond* http://www.inf.ed.ac.uk/teaching/courses/propm/papers/main_lan1.pdf

[7] Roberto Sebastiani *Efficient SAT Solving* http://www.cs.umd.edu/~micinski/posts/2012-09-22-efficient-sat-solving.html

[8] Mona Safar, M. Shalan, M. Watheq El-Kharashi, Ashraf Salem *Hardware Based Algorithm for Conflict Diagnosis in SAT Solver* http://www.cs.umd.edu/~micinski/posts/2012-09-22-efficient-sat-solving.html