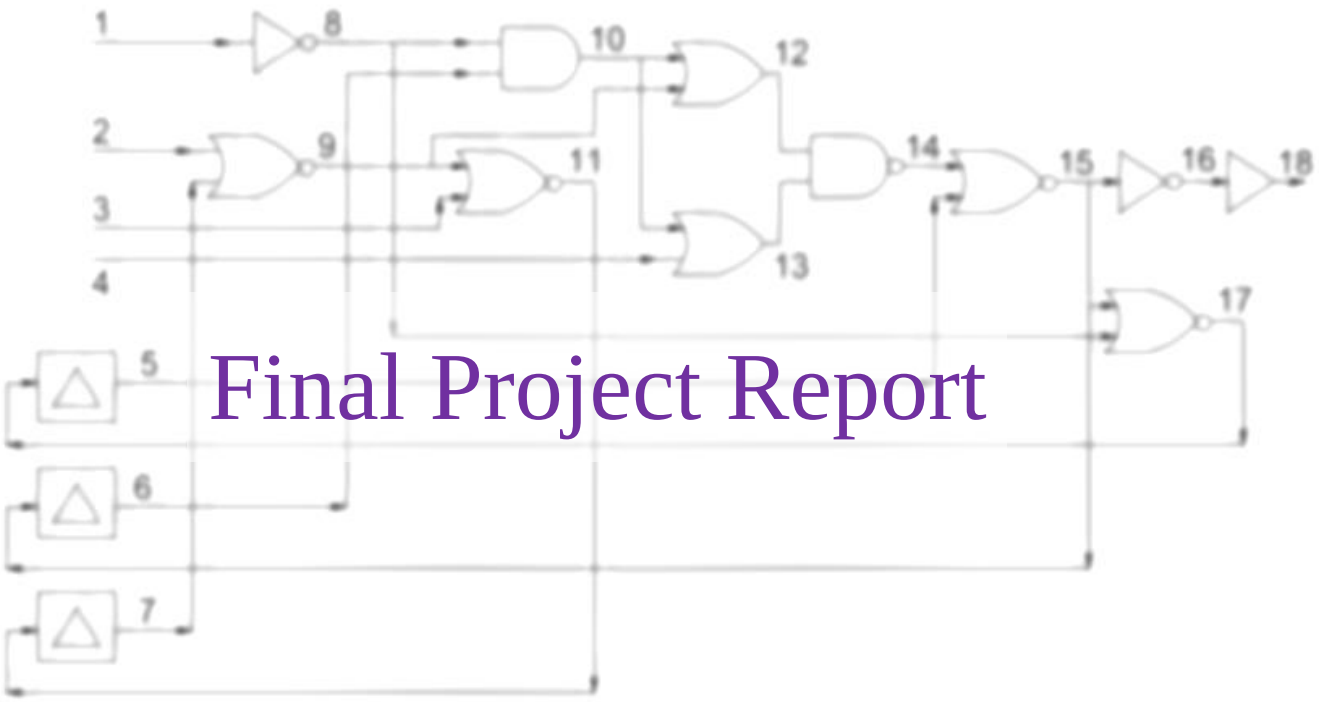
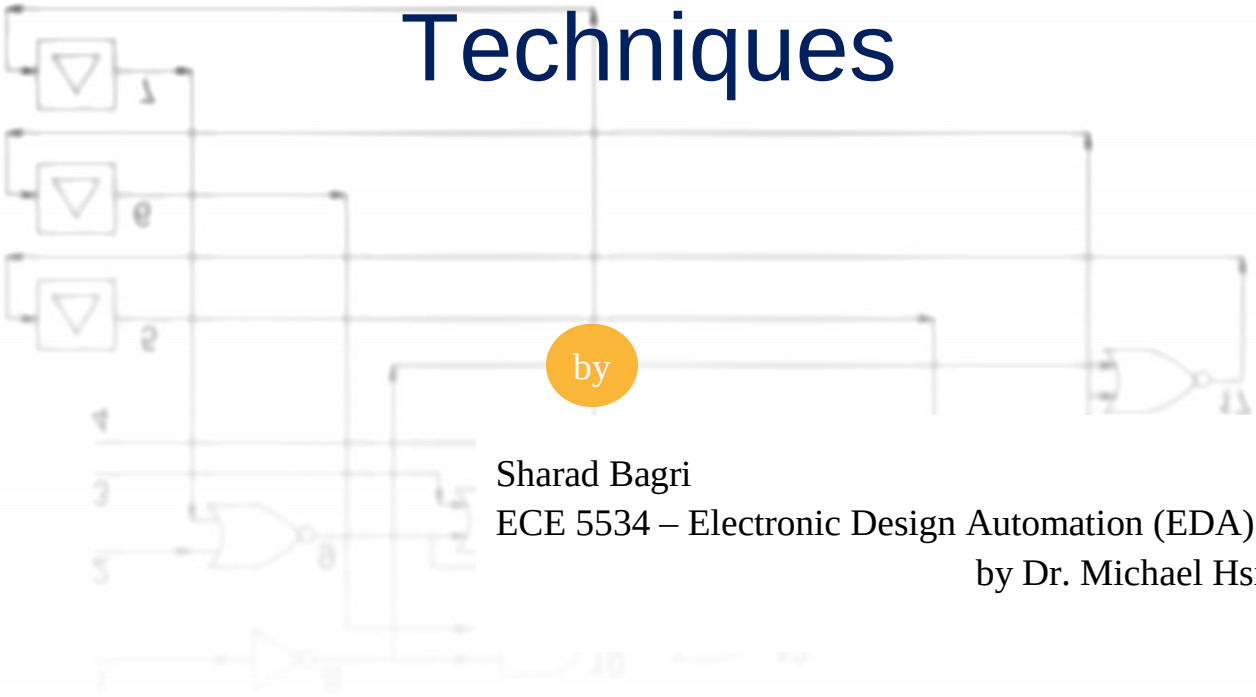


Final Project Report

Validation of RTL Circuits Using Deterministic Techniques



by

Sharad Bagri

ECE 5534 – Electronic Design Automation (EDA)

by Dr. Michael Hsiao

1. INTRODUCTION

Validation of circuits are becoming complex and time consuming as more and more transistors are fit onto the chip. Various ways are being studied on how to aid the process of validation of circuit in minimum time. In design process of circuits, first the behavioral model of the circuit is written in RTL languages. They are transformed to lower and lower level languages closer to the hardware as the project progresses. Transistor level details can be used to validate the circuits but it is generally less efficient as compared to validation vectors designed with help of higher level models. This happens because at transistor level, lot of high level details like Controllability Don't Care (CDC), Observability Don't Care (ODC), etc. are lost.

Vectors are required to validate the circuits. For large circuits simulation based approach in generating vectors generally works better than getting vectors through deterministic techniques. In deterministic techniques search space explodes just with few time frames unrolling of the circuit and so the mathematical model derived becomes too complex to be solved by conventional methods.

However, some states of the circuits are not reached with simulation based techniques. There are primarily two kinds of states which can't be reached.

1) Default cases put at RTL level design just for catching errors. They are not expected to reach when circuit has no error.
2) Some difficult to reach states which requires a specific set of conditions to be hit for many cycles. For example, setting of MSB of 32 bit counter when increments of the counter happen in steps of one and counting starts from zero.

In both the above cases, deterministic method can help a lot. It case 1, if the circuit is unrolled for a small number of cycles and its model is proved unsatisfiable then simulation technique can stop trying to reach there. In case 2, similar solution can be used to guide which conditions should be hit more.

2. PROJECT SPECIFIC BACKGROUND

This project required few third party software packages. They are

1) Z3 in Python: z3 is an open source tool developed by Microsoft. It is a Satisfiability Modulo Theory (SMT) solver. It is used in this project to see if a particular clause can be satisfied or not. Not all expression of C++ can be directly given to z3 and so transcoding needs to be done for them. For this purpose a C code parser implemented in python is used.
2) PyCParser: PyCParser in an open source C language parser written in python. It is maintained and authored by Eli Bendersky. It is designed to integrate into applications that require parsing of C code. In current project it is used to transcode C expressions into z3 expressions and so the libraries are hacked to suit the needs.
3) Python 2.7 was used to write the program as z3 is available only for python 2.7 and not on latest version of python which is 3.3. Initial development was done in python 3.3
4) Verilator: Verilator is an open source tool written by Wilson Snyder, Duane Galbi, Paul Wasson. It is used to covert VHDL code to C++ code. It has been in use for quite some time now and its output are always correct.

3. PSEUDO CODE

Get .h, .cpp file of the circuit and get numbers of coverage point to be checked from user

Read both files in separate lists.

Extract "`void Vtop::_sequent__TOP__1(Vtop__Syms* __restrict vlSymsp)`" function contents from cpp file and all signal names from .h file

Initialize parser and generator modules

For each coverage number

- Get all conditions which should be true and conditions which should be false
- Get condition just above the coverage number and determine if it needs to be true or false (1)
  - Find Signals involved in condition (1) → (2)
- Find all assignments and coverage number of signals in (2) → (3)
- Find new signals in (3) → (4)
- Find all assignments and coverage number for new signal in (4) → (5)
- Find new signals in (5) and add it 'all signals found' list → (6)
- For all signals found till (6)
  - Create bitvectors of it in z3 format
- For each Conditions involving only signals in list (6)
  - Check if it needs to be true or false and send it to C Parser → (7)
    - -Make AST of the condition in (7) → (8)
    - -Transcode AST into z3 format → (9)
  - Add the constraint in (9) in z3 solver
- For each assignment in (3)
  - Transcode assignment as a constraint in z3 format
  - Add it to stack of z3 conditions → (10)
  - Solve it and add the coverage number in sat or unsat list based on solution
    - --For each assignment in (5)
      - --Transcode assignment as a constraint in z3 format
      - --Add it to stack of z3 conditions → (11)
      - --Solve it and add the coverage number in sat or unsat list based on solution

----Pop out condition put in (11)

--Pop out condition put in (10)

-Print sat and unsat list

END

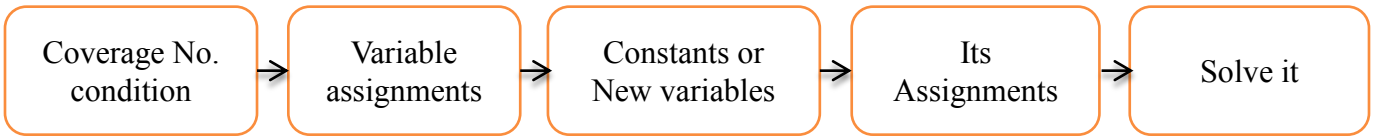


Figure 1: High level flow diagram of code

4. LEARNINGS AND DISCOVERIES

This project deepened my understanding of compilers and Abstract Syntax Tree (AST). I was unaware that first compilers read the program character by character and then tokenize it. Those tokens are similar to assembly code kind of program. These tokens can be used for transcoding program in one language to another language. As I discovered that many C++ expressions are not recognized by z3, use of AST in this case became evident. I searched and partly worked on tools to get AST. This exposed me to lex and yacc. Finally I decided to use pycparser as it was written in python and looked easier to use. Theoretically, AST can be made for full program and that can be used to transcode particular expressions but that would have been computationally costly. So, converting to AST was limited to few single line expressions only.

I intended to go two time frames deep in the circuit by going to two level deep assignments. However, there are local versions of signals present for some variables. So getting to those local variables was one level deep. So, if any local versions of the signals are not present then program goes back till two time frames. The assignments to local signals happen at the beginning or end of the function which simulates one cycle. There is no coverage number associated with these

locations. In case any assignment is found at such places then their coverage number is put as -1 so that it doesn't conflict with other coverage numbers and also provides valuable information of where the assignment happens.

Unrolling of the circuit is different when looked from gate level file perspective and from RTL level file perspective. In gate level, the circuit is a directly acyclic graph. So, moving back on the connections from output to input till some Primary Inputs (PI) or pseudo primary inputs (PPI) are reached gives component of one time frame. While in case of RTL circuits we need to track back the assignment of signal under consideration until it is being assigned by some input or register. Though both at RTL and gate level the meaning of unrolling remains same but there is a vast difference in implementation. With gate level description program can be made easily to make a graph and traverse it as it is just like another DAG. With RTL level description, connections have to be searched by going to each assignment statement and deciding which assignment involves the signal of interest.

For many circuits and most of their coverage's, good information can be found by looking at just the last two time frames. It is possible because normal circuits have very few states which are hard to reach.

For non-reachable conditions going deep doesn't explode and terminates just after few cycles. It is so because all the unreachable condition doesn't depend on a chain of variables. Normally, they are 'default' cases of assignment to one of the signals in Hardware Description Languages (HDLs). They are just put to alert the programmer if some mistake happened. In normal working condition they are never reached.

For normal registers and signals, search space may explode very soon as they may depend on a chain of variables. In those cases, going deeper becomes infeasible. The mathematical model generated for such variables becomes huge and practically unsolvable. In the maximal case, it may become as big as the program itself. However, if a state is very hard to reach then deterministic solutions can give concrete coverage points which would definitely not satisfy the condition. These 'Hard to Reach' states are hard to reach because typically because they require continually executing selected branches of code among all possible branches. This becomes tough when probabilistic techniques are used to find solution as they may try to hit more diverse set of branches in order to try more to reach the solution. However unsatisfiable assignments can normally be found with few time frame unrolling. As few branches actually can satisfy those constraints to reach 'hard to reach' states so it is easy to find out those constraints which are unsatisfiable with simply unrolling the circuit a few time frames. This helps in reducing the search space and can be used to guide probabilistic simulations to guide them to reach solution.

5. WORKIG OF THE CODE AND LEARNINGS FROM IT

RTL code mostly implement state machines which in programming terms are like switch-case

Figure 2: RTL code and corresponding C++ code

```
always @
(posedge clock
or posedge reset)
   begin : process_1
   if (reset ==  1'b 1)
      begin
      outp <= 1'b 0;
      end
   else
      begin
      outp <= 1'b 1;
      end
   end
endmodule
```

```
void Vtop::_sequent__TOP__1
(Vtop__Syms* __restrict vlSymsp)
{
    Vtop* __restrict
    vlTOPp VL_ATTR_UNUSED = vlSymsp->TOPp;
    // Body
    // ALWAYS at top.v:45
    if (vlTOPp->reset)
    {
        ++(vlSymsp->__Vcoverage[0]);
        vlTOPp->outp = 0;
    }
    else
    {
        ++(vlSymsp->__Vcoverage[1]);
        vlTOPp->outp = 1;
    }
}
```

block as show in the figure above. Verilator generated C++ code breaks down those cases to if-else blocks. All the successive if's go inside else of previous block. All assignments happen inside such blocks. It typically inserts coverage points as a first statement inside the branch. This design of the code can be used for a simplistic parsing of the condition. Code written for this project first searches for the coverage line by going to each line of the function simulating one cycle. When that line is found, it designates that code block as active code block and searches for if or else condition just above that block. If the condition is an 'if' condition then that condition is put in list of conditions which should be true. If the condition is 'else' then 'if' condition just above it is searched and that condition is put in the list of conditions which should be false. This way the code searches all the way till the beginning of function.

Once all the true and false conditions are found, signals involved in them are extracted. All assignments to those signals are found by parsing the full code again. Pattern like "particular Signal =" is searched in the full function and saved in a list referred as assignment1 henceforth.

All new signals in assignment1 are found and their corresponding assignments are saved in list referred as assignment2 henceforth.

In next solver is called. First all the condition on the signal just above coverage number line is put in the solver. Thereafter one assignment from assignment1 and one more from assignment2 is put in the solver. All the possible combination of assignment1 and assignment2 are tried one by one and it is checked whether any of them satisfy the constraints to reach that coverage line.

Coverage number of any such solution which satisfies those constraints is put in the Satisfiable assignments and those which don't satisfy are put in Unsatisfiable list.

6. RESULTS

The code was tested for circuits' b01, b06, b07 and b12. It worked fine for all of them and is expected to work fine for other circuits as well. The programs were run on laptop having Intel i5-3210 processor @ 2.5 GHz with 8.00 GB RAM with Windows 7 OS.

Circuit Name	Total Coverage point	Approximate Running Time (s)
b01	26	2
b06	25	2
b07	20	2
b12	105	10

It worked for all simple assignments. Some conditions which involved macros or arrays of bitvectors couldn't be solved. It is because they can't be dealt with just text parsing. They require compiler level knowledge of the whole program to make sense from them which is outside the scope of this project.

Output for coverage no 15 of circuit b01 is as follows

```
Coverage No. to test : 15
Coverage 15 requires (2 == (IData)(vlTOPp->v__DOT__process_1_stato)) to be true
It involves Signal : ['v__DOT__process_1_stato']
Which is assigned values at {'v__DOT__process_1_stato = 5;': 10,
'v__DOT__process_1_stato = 6;': 17, 'v__DOT__process_1_stato = 7;': 16,
'v__DOT__process_1_stato = 2;': 11, 'v__DOT__process_1_stato = 3;': 22,
'v__DOT__process_1_stato = 4;': 4, 'v__DOT__process_1_stato = 0;': 23,
'v__DOT__process_1_stato = 1;': 5}
Final list of signals on which coverage 15 depends ['v__DOT__process_1_stato']

All constraints on final signals are
(2 == (IData)(vlTOPp->v__DOT__process_1_stato)) should be true
(((((((0 == (IData)(vlTOPp->v__DOT__process_1_stato)) | (3 == (IData)(vlTOPp-
>v__DOT__process_1_stato))) | (1 == (IData)(vlTOPp->v__DOT__process_1_stato))) | (4
== (IData)(vlTOPp->v__DOT__process_1_stato))) | (2 == (IData)(vlTOPp-
>v__DOT__process_1_stato))) | (5 == (IData)(vlTOPp->v__DOT__process_1_stato))) | (6
== (IData)(vlTOPp->v__DOT__process_1_stato))) | (7 == (IData)(vlTOPp-
>v__DOT__process_1_stato))) should be true
(4 == (IData)(vlTOPp->v__DOT__process_1_stato)) should be false
(1 == (IData)(vlTOPp->v__DOT__process_1_stato)) should be false
```

(3 == (IData)(vlTOPp->v__DOT__process_1_stato)) should be false
(0 == (IData)(vlTOPp->v__DOT__process_1_stato)) should be false
Assignments 1 level deep and their coverage no. are {'v__DOT__process_1_stato = 5;': 10,
'v__DOT__process_1_stato = 6;': 17, 'v__DOT__process_1_stato = 7;': 16,
'v__DOT__process_1_stato = 2;': 11, 'v__DOT__process_1_stato = 3;': 22,
'v__DOT__process_1_stato = 4;': 4, 'v__DOT__process_1_stato = 0;': 23,
'v__DOT__process_1_stato = 1;': 5}
Assignments 2 level deep and their coverage no. are {}
Satisfiable Coverages are [11]
Unsatisfiable Coverages are [10, 17, 16, 22, 4, 23, 5]

CONCLUSIONS AND FUTURE WORK

Building this solver was a great learning experience. It helped understand concepts related to RTL designs and validation of RTL circuits better.

In future, circuit unrolling can be increased from 2 to more time frames. This would help in getting better sense of whether a condition is satisfiable or not.

This program can be integrated with some probabilistic simulation based solver to guide the braches to be taken in order to reach a particular coverage.