Symbolic simulator

History

This was promising technology back then, but somehow never caught on.
I met it when working in a processor design group (Amazing team by the way). They used it to verify cache transistor net-lists.
Over the years, i heard about several companies trying to sell EDA software of symbolic simulator. Are they still there?
Anyway, this algorithm still looks intriguing. Or maybe it is a hammer, looking for it's nail.

Introduction

Regular verilog simulator assigns to nets one of 4 values: z, x, 0, 1
There are also drive strengths, but for RTL code this is immaterial.

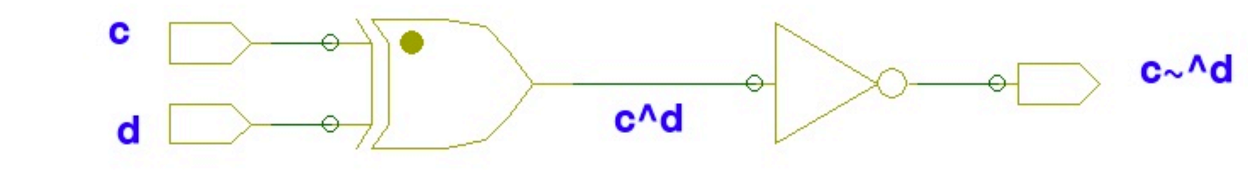
Symbolic simulator allows adding more. It allows a net to be driven by a symbol.
Usually symbol is just a letter, let say "**a**". We tell the simulator, that we will use symbol "**a**" as a legal value driven on any net.
Then we tell the simulator to drive symbol "**a**" to a net "inx". Effectively, with one symbol we are running 2 simulations at the same time.

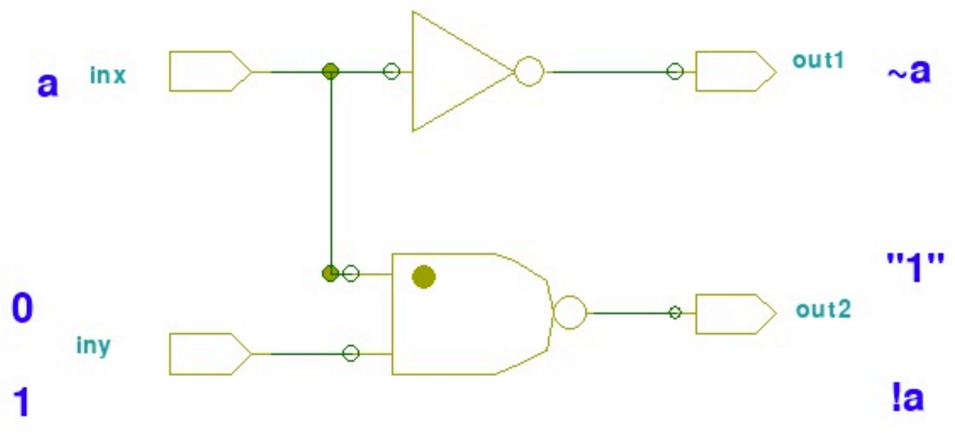
To help us manage multiple combinatorial expressions, we employ BDD package . Next paragraph gives some background on BDDs.

If we drive input of the inverter to "**a**", then the output of the inverter, becomes **"!a"** - inverted **"a"**.   **"!a"** is an expression kept in BDD class. For simulator, expression is an integer index into the BDD class instance. BDD is family of algorithms that enable manipulation of boolean formulas and making them canonic in form (Same formula always has the same representation).

On "iny" input we drive "0" followed by "1". The nand toggles between "1" when "iny" == 0 and "**!a**" when "iny" is one.

That is the gist of it. Look at XOR for more insight.



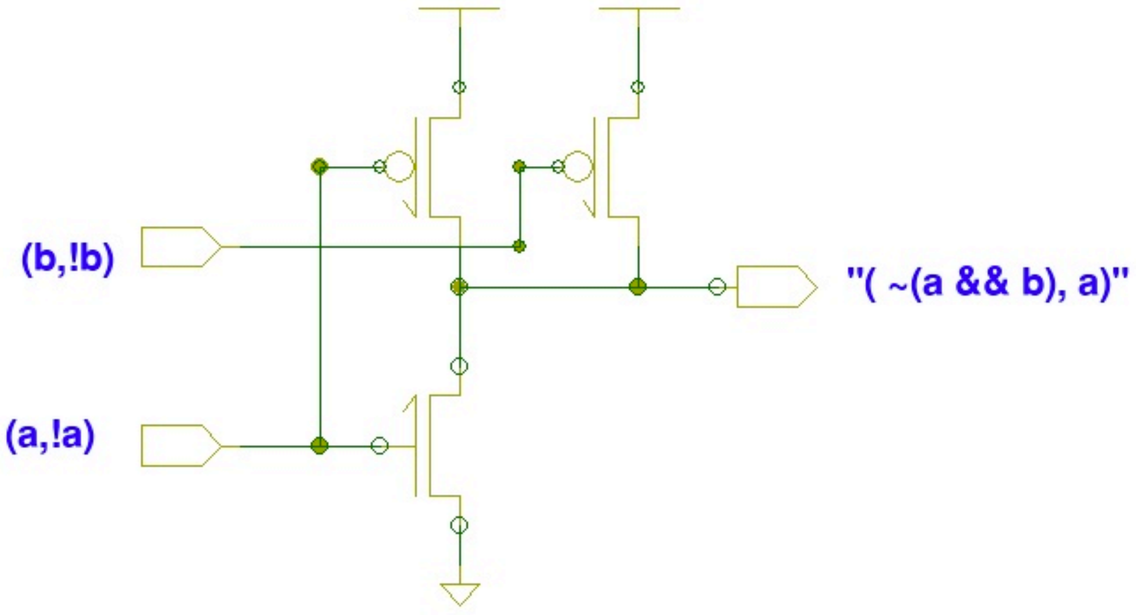


Transistor, including bidirs

For transistor network each node gets two expressions.

One for condition of it to become "1" and one for condition to be driven to "0".

As example, top input driven to symbol "b", and bottom input to "a". "b" is condition for this to be driven to "1" and "!b" to be driven to zero. The output gets more complicated formula, but notice that it will catch collision, when a==1 and b==0.



If number and position of symbols are carefully chosen, then in a transistor networks, this simulation can catch horrible errors of collisions and also simulate transistor forests, with more accuracy than vanilla verilog simulator with nmos, pmos and cmos.

End of theory. You can google "symbolic verilog simulator" for more. And by the way, if we declare all inputs to be driven by symbols, we actually getting to formal verification. Unfortunately this will lead to the same zombi apocalypse as mainstream formal (meant to say combi-natorial). So "symbolic" may be viewed as kinda poor man's formal. Or smart person's formal.

Keeping the expressions - BDD

To represent combinatorial expressions, like (a && !b). i am using BDD package connected to python. The one currently in use is '**dd**' (Pip it).

Bdd stands for Binary Decision Diagram. There are various optimized versions, which usually add a letter to BDD, like OBDD which stands for ordered BDD. Order makes canonical representation simpler.

"dd" 's ' documentation is way too academic for me, filled with irrelevant smarts. But 3 functions that added on top of that are :

Cx = combineOr(Ax,Bx)

Cx = combineAnd(Ax,Bx)

Cx = invert(Ax)

Where Ax,Bx,Cx are all object instances of expressions from relevant BDD class. Just two more functions clinch the deal:

Ax = getObj('tb.dut.enable').   : where, the value of net enable is peeked and returns the object from the bdd package which represents the expression.

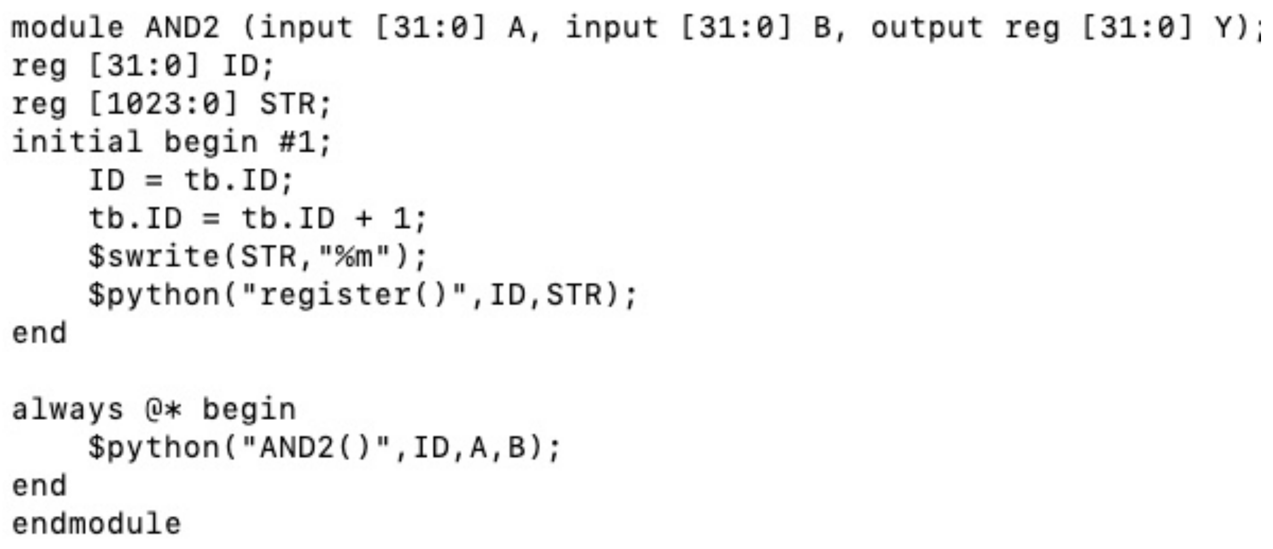
forcify(Path,Cx). : this function translates the object into a number, which is deposited on the correct net. 32bits is way too much, but i don't care.

My toy simulator.

I was lazy. Instead of creating RTL to SymbolRTL translator, I did RTL synthesis to simple liberty library (INV,BUF, NAND2, ...). The resulting gate-level went through my python мясорубка (meat-mincer - this is to brag my mastery of 3 human languages. On the other hand my mother spoke freely 5. Bragging fades away...) . This app reads verilog code and creates data structure of the code it read. Then python functions can modify the structure. In the end, it may dump changed verilog back. In current case, Python module splits all busses to individual bits and makes each wire 32 bits wide integer. Integer, because it allows to represent pointers to BDD class. Then second module reads top level and creates test-bench verilog file. (Both are in pyver/pybin3)

Finally, Verilog simulator is run (any kind, free or licensed) , driven by P-D-V (python driven verification), where all standard cells are zombies manipulated by a python code.

The verilog representation of a simple library zombi cell looks like :

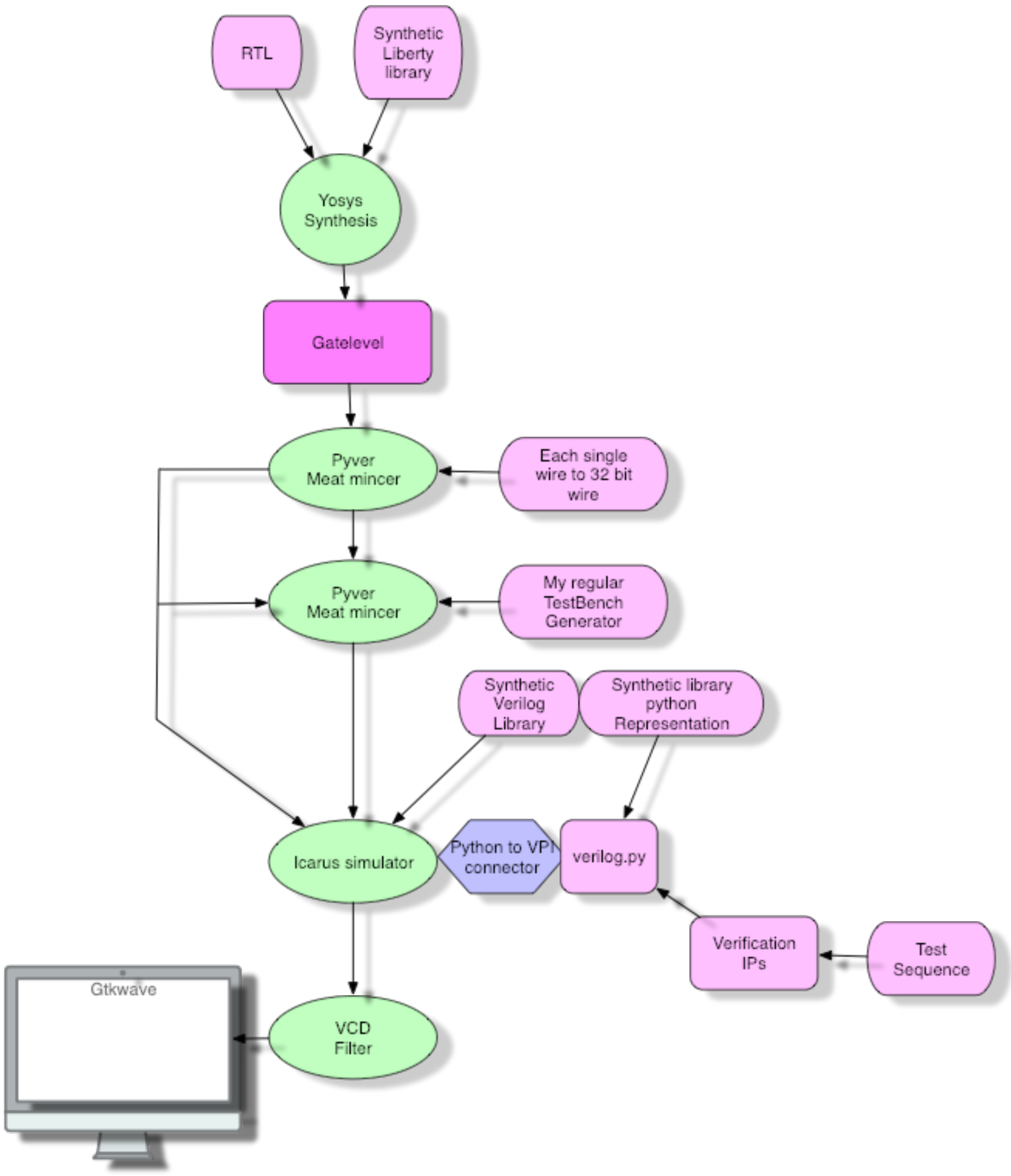


```
module AND2 (input [31:0] A, input [31:0] B, output reg [31:0] Y);
reg [31:0] ID;
reg [1023:0] STR;
initial begin #1;
    ID = tb.ID;
    tb.ID = tb.ID + 1;
    $swrite(STR,"%m");
    $python("register()",ID,STR);
end

always @* begin
    $python("AND2()",ID,A,B);
end
endmodule
```

You can see that the cell (and all others) gathers inputs and then calls a python function by the same name. Part of the call is unique ID which is used to keep full path to that specific instance. Based on the path, python function can force the relevant "Y" output to correct value.

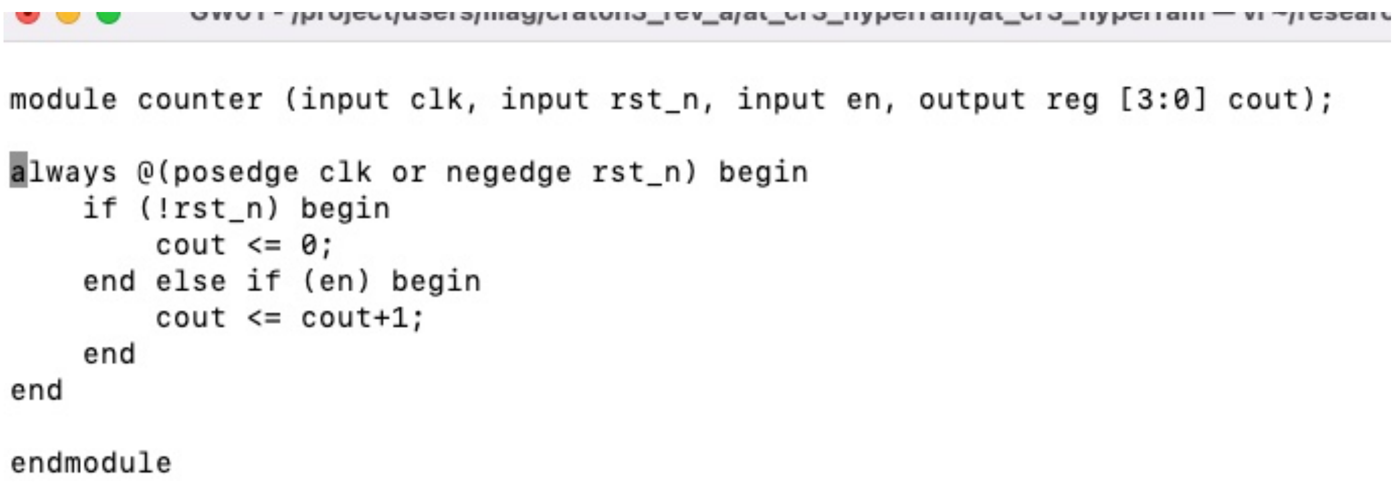
The full flow:



`Test sequence` is my regular sequence file used in Python-Driven-Verification (PDV not cocoTb). It is read by "sequence.py" imported into verilog.py The simulation is run and VCD file produced. But here is the snag = the 32 bit values in the VCD are random integers assigned by BDD package to various expression. To make sense of it, VCD filter app reads the VCD and translates each 32bit value to a string that represents the expression.

Then "gtkwave" can display sensible stuff. It has display "ascii" option.

Here is a simple rtl:



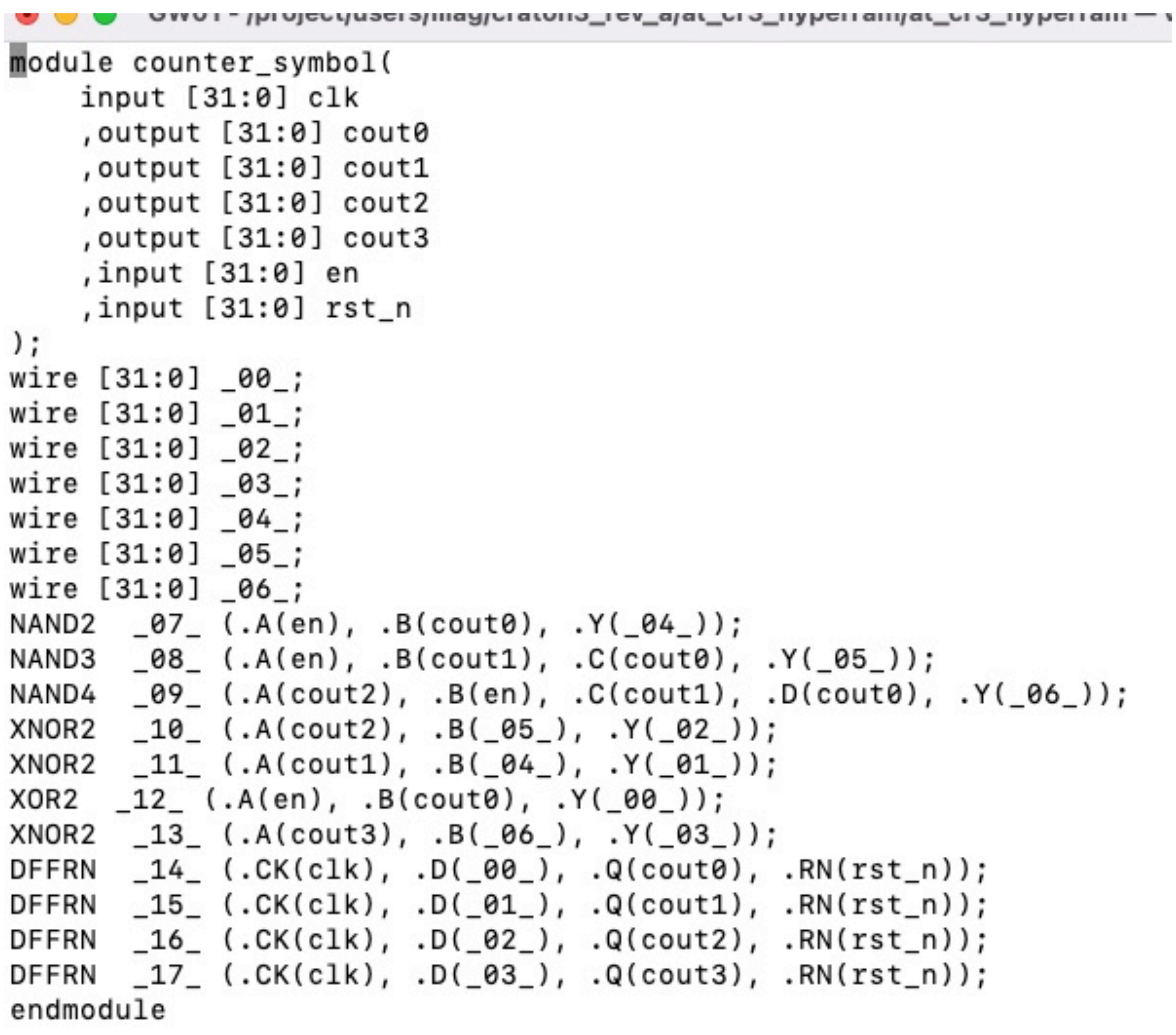
```
module counter (input clk, input rst_n, input en, output reg [3:0] cout);

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        cout <= 0;
    end else if (en) begin
        cout <= cout+1;
    end
end

endmodule
```

This is just a counter. What do You think will happen if some time after reset, we force symbol "a" on input "en"?

Here is the actual code that went into the simulator (synthesized and expanded):



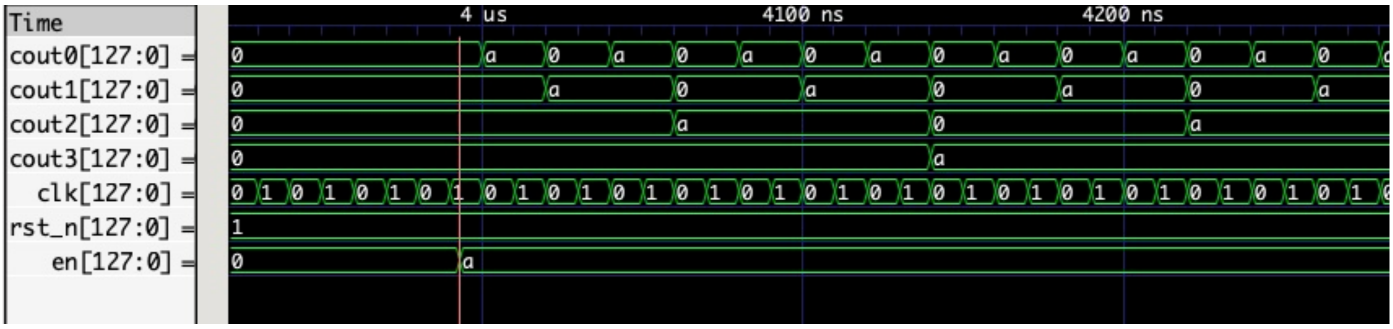
```
module counter_symbol(
    input [31:0] clk
    ,output [31:0] cout0
    ,output [31:0] cout1
    ,output [31:0] cout2
    ,output [31:0] cout3
    ,input [31:0] en
    ,input [31:0] rst_n
);
wire [31:0] _00_;
wire [31:0] _01_;
wire [31:0] _02_;
wire [31:0] _03_;
wire [31:0] _04_;
wire [31:0] _05_;
wire [31:0] _06_;
NAND2  _07_ (.A(en), .B(cout0), .Y(_04_));
NAND3  _08_ (.A(en), .B(cout1), .C(cout0), .Y(_05_));
NAND4  _09_ (.A(cout2), .B(en), .C(cout1), .D(cout0), .Y(_06_));
XNOR2  _10_ (.A(cout2), .B(_05_), .Y(_02_));
XNOR2  _11_ (.A(cout1), .B(_04_), .Y(_01_));
XOR2  _12_ (.A(en), .B(cout0), .Y(_00_));
XNOR2  _13_ (.A(cout3), .B(_06_), .Y(_03_));
DFFRN  _14_ (.CK(clk), .D(_00_), .Q(cout0), .RN(rst_n));
DFFRN  _15_ (.CK(clk), .D(_01_), .Q(cout1), .RN(rst_n));
DFFRN  _16_ (.CK(clk), .D(_02_), .Q(cout2), .RN(rst_n));
DFFRN  _17_ (.CK(clk), .D(_03_), .Q(cout3), .RN(rst_n));
endmodule
```

Keep in mind that all "standart cells" are driven by python from within. By something like this:

```
def OR3(Id,A,B,C):
    Path,A,B,C = opening3(Id,A,B,C)
    if not Path: return
    One = combineOr(C,combineOr(A,B))
    forcing(Path,One)
```

`Opening` translates the parameters this function was called with to valid Path and BDD expression objects. `Combines` calculate the correct output expression. `Forcing` deposits the value (index of the expression object) onto the verilog net.

Here is trace of waves, after VCD was filtered :



You can see that once (**en**)able becomes **"a"** - the counter counts.

Sofar

Is this interesting example? No.   Are there bugs in my current flow? Most probably Yes.

What are the examples where symbolic simulation shines? What is the "**NAIL**" application? Wait for the next installement.

Can You do it in SV+UVM? Maybe, but i dont condone suicidal tendencies.

This flow is not in vlsistuff repository yet. It will be there eventually. When it will be more solid.

But You are allowed to beg for it. 😢