Yosys Application Note 011:
Interactive Design Investigation

Clifford Wolf

Original Version December 2013

Abstract—**Yosys [1] can be a great environment for building custom synthesis flows. It can also be an excellent tool for teaching and learning Verilog based RTL synthesis. In both applications it is of great importance to be able to analyze the designs it produces easily.**

This Yosys application note covers the generation of circuit diagrams with the Yosys show **command, the selection of interesting parts of the circuit using the** select **command, and briefly discusses advanced investigation commands for evaluating circuits and solving SAT problems.**

I. Installation and Prerequisites

This Application Note is based on the Yosys [1] GIT Rev. 2b90ba1 from 2013-12-08. The README file covers how to install Yosys. The show command requires a working installation of GraphViz [2] and [3] for generating the actual circuit diagrams.

II. Overview

This application note is structured as follows:

Sec. III introduces the show command and explains the symbols used in the circuit diagrams generated by it.

Sec. IV introduces additional commands used to navigate in the design, select portions of the design, and print additional information on the elements in the design that are not contained in the circuit diagrams.

Sec. V introduces commands to evaluate the design and solve SAT problems within the design.

Sec. VI concludes the document and summarizes the key points.

III. Introduction to the show command

The show command generates a circuit diagram for the design in its current state. Various options can be used to change the appearance of the circuit diagram, set the name and format for the output file, and so forth. When called without any special options, it saves the circuit diagram in a temporary file and launches xdot to display the diagram. Subsequent calls to show re-use the xdot instance (if still running).

```
1   $ cat example.ys
2   read_verilog example.v
3   show -pause
4   proc
5   show -pause
6   opt
7   show -pause
8
9   $ cat example.v
10  module example(input clk, a, b, c,
11                  output reg [1:0] y);
12      always @(posedge clk)
13          if (c)
14              y <= c ? a + b : 2'd0;
15  endmodule
```

Figure 1. Yosys script with show commands and example design

A. A simple circuit

Fig. 1 shows a simple synthesis script and a Verilog file that demonstrate the usage of show in a simple setting. Note that show is called with the -pause option, that halts execution of the Yosys script until the user presses the Enter key. The show -pause command also allows the user to enter an interactive shell to further investigate the circuit before continuing synthesis.

So this script, when executed, will show the design after each of the three synthesis commands. The generated circuit diagrams are shown in Fig. 2.

The first diagram (from top to bottom) shows the design directly after being read by the Verilog front-end. Input and output ports are displayed as octagonal shapes. Cells are displayed as rectangles with inputs on the left and outputs on the right side. The cell labels are two lines long: The first line contains a unique identifier for the cell and the second line contains the cell type. Internal cell types are prefixed with a dollar sign. The Yosys manual contains a chapter on the internal cell library used in Yosys.

Constants are shown as ellipses with the constant value as label. The syntax <bit_width>'<bits> is used for for constants that are not 32-bit wide and/or contain bits that are not 0 or 1 (i.e. x or z). Ordinary 32-bit constants are written using decimal numbers.

Single-bit signals are shown as thin arrows pointing from the driver to the load. Signals that are multiple bits wide are shown as think arrows.

Finally *processes* are shown in boxes with round corners. Processes are Yosys' internal representation of the decision-trees and synchronization events modelled in a Verilog always-block. The label reads PROC followed by a unique identifier in the first line and contains the source code location of the original always-block in the 2nd line. Note how the multiplexer from the ?:-expression
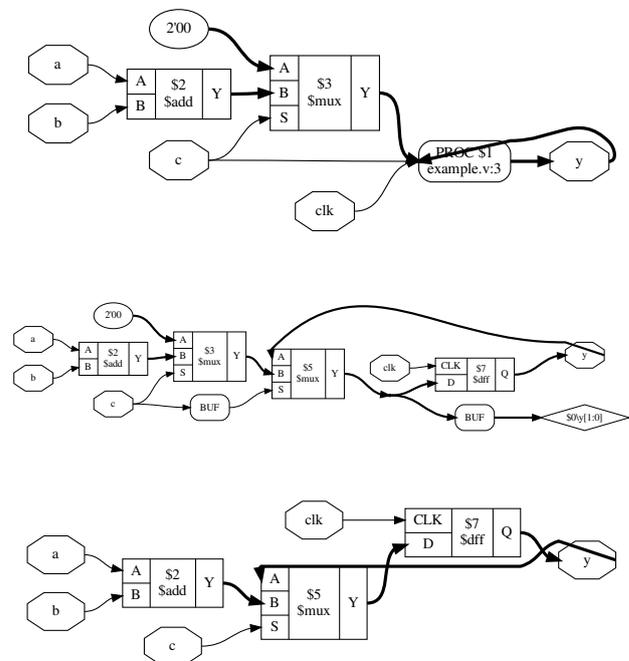

Figure 2. Output of the three show commands from Fig. 1

is represented as a $mux cell but the multiplexer from the if-statement is yet still hidden within the process.

The proc command transforms the process from the first diagram into a multiplexer and a d-type flip-flip, which brings us to the 2nd diagram.

The Rhombus shape to the right is a dangling wire. (Wire nodes are only shown if they are dangling or have "public" names, for example names assigned from the Verilog input.) Also note that the design now contains two instances of a BUF-node. This are artefacts left behind by the proc-command. It is quite usual to see such artefacts after calling commands that perform changes in the design, as most commands only care about doing the transformation in the least complicated way, not about cleaning up after them. The next call to clean (or opt, which includes clean as one of its operations) will clean up this artefacts. This operation is so common in Yosys scripts that it can simply be abbreviated with the ;; token, which doubles as separator for commands. Unless one wants to specifically analyze this artefacts left behind some operations, it is therefore recommended to always call clean before calling show.

In this script we directly call opt as next step, which finally leads us to the 3rd diagram in Fig. 2. Here we see that the opt command not only has removed the artifacts left behind by proc, but also determined correctly that it can remove the first $mux cell without changing the behavior of the circuit.

B. Break-out boxes for signal vectors

As has been indicated by the last example, Yosys is can manage signal vectors (aka. multi-bit wires or buses) as native objects. This provides great advantages when analyzing circuits that operate on wide integers. But it also introduces some additional complexity
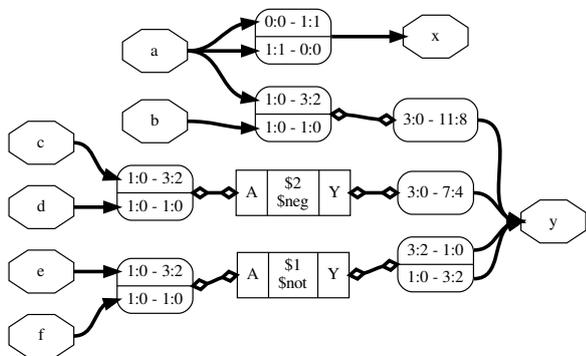

Figure 3. Output of yosys -p 'proc; opt; show' splice.v

```
1  module splice_demo(a, b, c, d, e, f, x, y);
2
3  input [1:0] a, b, c, d, e, f;
4  output [1:0] x = {a[0], a[1]};
5
6  output [11:0] y;
7  assign {y[11:4], y[1:0], y[3:2]} =
8                  {a, b, -{c, d}, ~{e, f}};
9
10 endmodule
```

Figure 4. splice.v
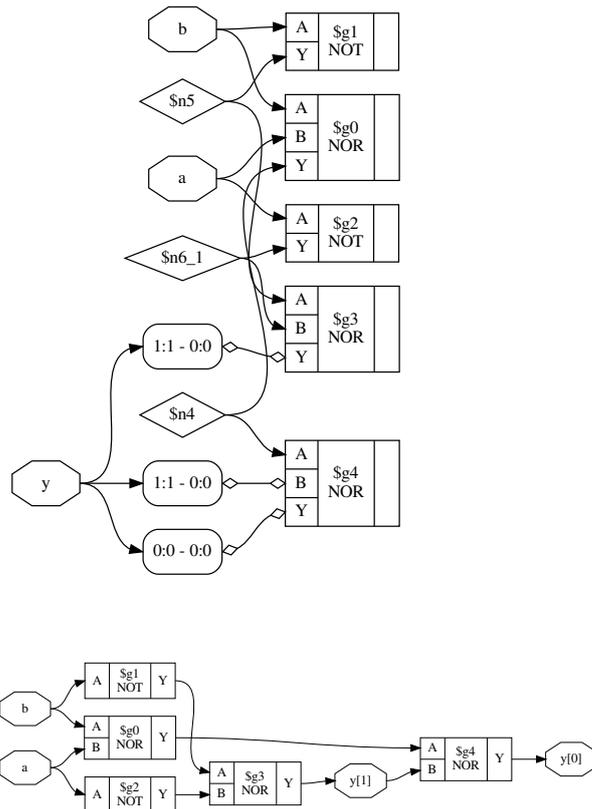
Figure 5. Effects of splitnets command and of providing a cell library. (The circuit is a half-adder built from simple CMOS gates.)

when the individual bits of of a signal vector are accessed. The example show in Fig. 3 and 4 demonstrates how such circuits are visualized by the show command.

The key elements in understanding this circuit diagram are of course the boxes with round corners and rows labeled <MSB_LEFT>:<LSB_LEFT> – <MSB_RIGHT>:<LSB_RIGHT>. Each of this boxes has one signal per row on one side and a common signal for all rows on the other side. The <MSB>:<LSB> tuples specify which bits of the signals are broken out and connected. So the top row of the box connecting the signals a and x indicates that the bit 0 (i.e. the range 0:0) from signal a is connected to bit 1 (i.e. the range 1:1) of signal x.

Lines connecting such boxes together and lines connecting such boxes to cell ports have a slightly different look to emphasise that they are not actual signal wires but a necessity of the graphical representation. This distinction seems like a technicality, until one wants to debug a problem related to the way Yosys internally represents signal vectors, for example when writing custom Yosys commands.

C. Gate level netlists

Finally Fig. 5 shows two common pitfalls when working with designs mapped to a cell library. The top figure has two problems: First Yosys did not have access to the cell library when this diagram was generated, resulting in all cell ports defaulting to being inputs. This is why all ports are drawn on the left side the cells are

awkwardly arranged in a large column. Secondly the two-bit vector y requires breakout-boxes for its individual bits, resulting in an unnecessary complex diagram.

For the 2nd diagram Yosys has been given a description of the cell library as Verilog file containing blackbox modules. There are two ways to load cell descriptions into Yosys: First the Verilog file for the cell library can be passed directly to the `show` command using the `-lib <filename>` option. Secondly it is possible to load cell libraries into the design with the `read_verilog -lib <filename>` command. The 2nd method has the great advantage that the library only needs to be loaded once and can then be used in all subsequent calls to the `show` command.

In addition to that, the 2nd diagram was generated after `split-net -ports` was run on the design. This command splits all signal vectors into individual signal bits, which is often desirable when looking at gate-level circuits. The `-ports` option is required to also split module ports. Per default the command only operates on interior signals.

D. Miscellaneous notes

Per default the `show` command outputs a temporary `dot` file and launches `xdot` to display it. The options `-format`, `-viewer` and `-prefix` can be used to change format, viewer and filename prefix. Note that the `pdf` and `ps` format are the only formats that support plotting multiple modules in one run.

In densely connected circuits it is sometimes hard to keep track of the individual signal wires. For this cases it can be useful to call `show` with the `-colors <integer>` argument, which randomly assigns colors to the nets. The integer (> 0) is used as seed value for the random color assignments. Sometimes it is necessary it try some values to find an assignment of colors that looks good.

The command `help  show` prints a complete listing of all options supported by the `show` command.

IV. NAVIGATING THE DESIGN

Plotting circuit diagrams for entire modules in the design brings us only helps in simple cases. For complex modules the generated circuit diagrams are just stupidly big and are no help at all. In such cases one first has to select the relevant portions of the circuit.

In addition to *what* to display one also needs to carefully decide *when* to display it, with respect to the synthesis flow. In general it is a good idea to troubleshoot a circuit in the earliest state in which a problem can be reproduced. So if, for example, the internal state before calling the `techmap` command already fails to verify, it is better to troubleshoot the coarse-grain version of the circuit before `techmap` than the gate-level circuit after `techmap`.

Note: It is generally recommended to verify the internal state of a design by writing it to a Verilog file using `write_verilog -noexpr` and using the simulation models from `simlib.v` and `simcells.v` from the Yosys data directory (as printed by `yosys-config --datdir`).

A. Interactive Navigation

Once the right state within the synthesis flow for debugging the circuit has been identified, it is recommended to simply add the `shell` command to the matching place in the synthesis script. This command will stop the synthesis at the specified moment and go to shell mode, where the user can interactively enter commands.

For most cases, the shell will start with the whole design selected (i.e. when the synthesis script does not already narrow the selection). The command `ls` can now be used to create a list

```
1   yosys> ls
2
3   1 modules:
4     example
5
6   yosys> cd example
7
8   yosys [example]> ls
9
10  7 wires:
11    $0\y[1:0]
12    $add$example.v:5$2_Y
13    a
14    b
15    c
16    clk
17    y
18
19  3 cells:
20    $add$example.v:5$2
21    $procdff$7
22    $procmux$5
```

Figure 6. Demonstration of `ls` and `cd` using `example.v` from Fig. 1

of all modules. The command `cd` can be used to switch to one of the modules (type `cd  ..` to switch back). Now the `ls` command lists the objects within that module. Fig. 6 demonstrates this using the design from Fig. 1.

There is a thing to note in Fig. 6: We can see that the cell names from Fig. 2 are just abbreviations of the actual cell names, namely the part after the last dollar-sign. Most auto-generated names (the ones starting with a dollar sign) are rather long and contains some additional information on the origin of the named object. But in most cases those names can simply be abbreviated using the last part.

Usually all interactive work is done with one module selected using the `cd` command. But it is also possible to work from the design-context (`cd  ..`). In this case all object names must be prefixed with `<module_name>/`. For example `a*/b*` would refer to all objects whose names start with b from all modules whose names start with a.

The `dump` command can be used to print all information about an object. For example `dump  $2` will print Fig. 7. This can for example be useful to determine the names of nets connected to cells, as the net-names are usually suppressed in the circuit diagram if

```
1    attribute \src "example.v:5"
2    cell $add $add$example.v:5$2
3      parameter \A_SIGNED 0
4      parameter \A_WIDTH 1
5      parameter \B_SIGNED 0
6      parameter \B_WIDTH 1
7      parameter \Y_WIDTH 2
8      connect \A \a
9      connect \B \b
10     connect \Y $add$example.v:5$2_Y
11   end
```

Figure 7. Output of `dump  $2` using the design from Fig. 1 and Fig. 2
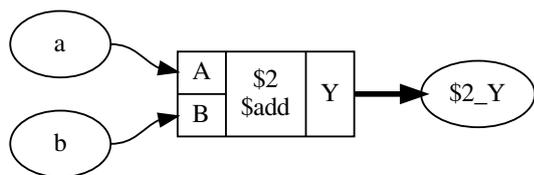
Figure 8. Output of show after select $2 or select t:$add (see also Fig. 2)

```
1  module sumprod(a, b, c, sum, prod);
2
3     input [7:0] a, b, c;
4     output [7:0] sum, prod;
5
6     {* sumstuff *}
7     assign sum = a + b + c;
8     {* *}
9
10    assign prod = a * b * c;
11
12 endmodule
```

Figure 10. Another test module for operations on selections

they are auto-generated.

For the remainder of this document we will assume that the commands are run from module-context and not design-context.

B. Working with selections

When a module is selected using the cd command, all commands (with a few exceptions, such as the read_* and write_* commands) operate only on the selected module. This can also be useful for synthesis scripts where different synthesis strategies should be applied to different modules in the design.

But for most interactive work we want to further narrow the set of selected objects. This can be done using the select command.

For example, if the command select $2 is executed, a subsequent show command will yield the diagram shown in Fig. 8. Note that the nets are now displayed in ellipses. This indicates that they are not selected, but only shown because the diagram contains a cell that is connected to the net. This of course makes no difference for the circuit that is shown, but it can be a useful information when manipulating selections.

Objects can not only be selected by their name but also by other properties. For example select t:$add will select all cells of type $add. In this case this is also yields the diagram shown in Fig. 8.

The output of help select contains a complete syntax reference for matching different properties.

Many commands can operate on explicit selections. For example the command dump t:$add will print information on all $add cells in the active module. Whenever a command has [selection] as last argument in its usage help, this means that it will use the engine behind the select command to evaluate additional arguments and use the resulting selection instead of the selection created by the last select command.

Normally the select command overwrites a previous selection. The commands select -add and select -del can be used to add or remove objects from the current selection.

The command select -clear can be used to reset the selection to the default, which is a complete selection of everything in the current module.

C. Operations on selections

The select command is actually much more powerful than it might seem on the first glimpse. When it is called with multiple arguments, each argument is evaluated and pushed separately on a stack. After all arguments have been processed it simply creates the union of all elements on the stack. So the following command will select all $add cells and all objects with the foo attribute set:

```
select t:$add a:foo
```

(Try this with the design shown in Fig. 9. Use the select -list command to list the current selection.)

In many cases simply adding more and more stuff to the selection is an ineffective way of selecting the interesting part of the design. Special arguments can be used to combine the elements on the stack. For example the %i arguments pops the last two elements from the stack, intersects them, and pushes the result back on the stack. So the following command will select all $add cells that have the foo attribute set:

```
1  module foobaraddsub(a, b, c, d, fa, fs, ba, bs);
2     input [7:0] a, b, c, d;
3     output [7:0] fa, fs, ba, bs;
4     assign fa = a + (* foo *) b;
5     assign fs = a - (* foo *) b;
6     assign ba = c + (* bar *) d;
7     assign bs = c - (* bar *) d;
8  endmodule
```

Figure 9. Test module for operations on selections
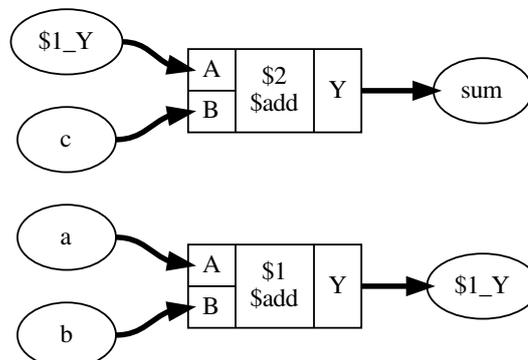

Figure 11. Output of show a:sumstuff on Fig. 10
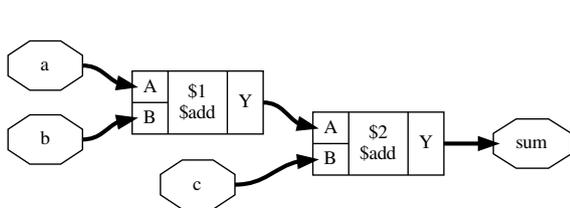
Figure 12. Output of `show a:sumstuff %x` on Fig. 10

```
select t:$add a:foo %i
```

The listing in Fig. 10 uses the Yosys non-standard `{* ... *}` syntax to set the attribute `sumstuff` on all cells generated by the first assign statement. (This works on arbitrary large blocks of Verilog code an can be used to mark portions of code for analysis.)

Selecting `a:sumstuff` in this module will yield the circuit diagram shown in Fig. 11. As only the cells themselves are selected, but not the temporary wire `$1_Y`, the two adders are shown as two disjunct parts. This can be very useful for global signals like clock and reset signals: just unselect them using a command such as `select -del clk rst` and each cell using them will get its own net label.

In this case however we would like to see the cells connected properly. This can be achieved using the `%x` action, that broadens the selection, i.e. for each selected wire it selects all cells connected to the wire and vice versa. So `show a:sumstuff %x` yields the diagram shown in Fig. 12.

D. Selecting logic cones

Fig. 12 shows what is called the *input cone* of `sum`, i.e. all cells and signals that are used to generate the signal `sum`. The `%ci` action can be used to select the input cones of all object in the top selection in the stack maintained by the `select` command.

As the `%x` action, this commands broadens the selection by one "step". But this time the operation only works against the direction of data flow. That means, wires only select cells via output ports and cells only select wires via input ports.

Fig. 13 show the sequence of diagrams generated by the following commands:

```
show prod
show prod %ci
show prod %ci %ci
show prod %ci %ci %ci
```

When selecting many levels of logic, repeating `%ci` over and over again can be a bit dull. So there is a shortcut for that: the number of iterations can be appended to the action. So for example the action `%ci3` is identical to performing the `%ci` action three times.

The action `%ci*` performs the `%ci` action over and over again until it has no effect anymore.

In most cases there are certain cell types and/or ports that should not be considered for the `%ci` action, or we only want to follow certain cell types and/or ports. This can be achieved using additional patterns that can be appended to the `%ci` action.

Lets consider the design from Fig. 14. It serves no purpose other than being a non-trivial circuit for demonstrating some of the advanced Yosys features. We synthesize the circuit using `proc`; `opt`; `memory`; `opt` and change to the `memdemo` module with `cd memdemo`. If we type `show` now we see the diagram shown in Fig. 15.

But maybe we are only interested in the tree of multiplexers that select the output value. In order to get there, we would start by just showing the output signal and its immediate predecessors:

Figure 13. Objects selected by `select prod %ci...`

```verilog
 1  module memdemo(clk, d, y);
 2
 3  input clk;
 4  input [3:0] d;
 5  output reg [3:0] y;
 6
 7  integer i;
 8  reg [1:0] s1, s2;
 9  reg [3:0] mem [0:3];
10
11  always @(posedge clk) begin
12      for (i = 0; i < 4; i = i+1)
13          mem[i] <= mem[(i+1) % 4] + mem[(i+2) % 4];
14      { s2, s1 } = d ? { s1, s2 } ^ d : 4'b0;
15      mem[s1] <= d;
16      y <= mem[s2];
17  end
18
19  endmodule
```

Figure 14. Demo circuit for demonstrating some advanced Yosys features
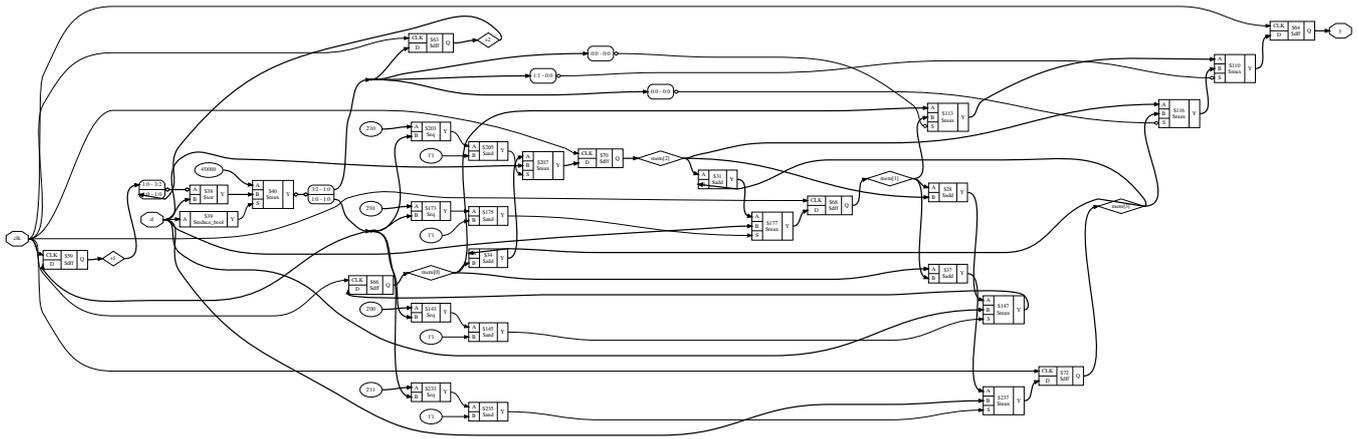
Figure 15. Complete circuit diagram for the design shown in Fig. 14

```
show y %ci2
```

From this we would learn that y is driven by a $dff cell, that y is connected to the output port Q, that the clk signal goes into the CLK input port of the cell, and that the data comes from a auto-generated wire into the input D of the flip-flop cell.

As we are not interested in the clock signal we add an additional pattern to the %ci action, that tells it to only follow ports Q and D of $dff cells:

```
show y %ci2:+$dff[Q,D]
```

To add a pattern we add a colon followed by the pattern to the %ci action. The pattern it self starts with – or +, indicating if it is an include or exclude pattern, followed by an optional comma separated list of cell types, followed by an optional comma separated list of port names in square brackets.

Since we know that the only cell considered in this case is a $dff cell, we could as well only specify the port names:

```
show y %ci2:+[Q,D]
```

Or we could decide to tell the %ci action to not follow the CLK input:

```
show y %ci2:-[CLK]
```

Next we would investigate the next logic level by adding another %ci2 to the command:
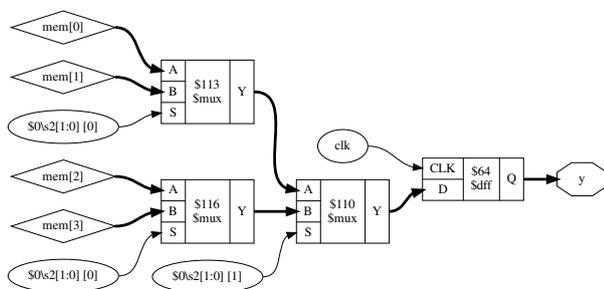
Figure 16. Output of show y %ci2:+$dff[Q,D] %ci*:-$mux[S]:-$dff

```
show y %ci2:-[CLK] %ci2
```

From this we would learn that the next cell is a $mux cell and we would add additional pattern to narrow the selection on the path we are interested. In the end we would end up with a command such as

```
show y %ci2:+$dff[Q,D] %ci*:-$mux[S]:-$dff
```

in which the first %ci jumps over the initial d-type flip-flop and the 2nd action selects the entire input cone without going over multiplexer select inputs and flip-flop cells. The diagram produces by this command is shown in Fig. 16.

Similar to %ci exists an action %co to select output cones that accepts the same syntax for pattern and repetition. The %x action mentioned previously also accepts this advanced syntax.

This actions for traversing the circuit graph, combined with the actions for boolean operations such as intersection (%i) and difference (%d) are powerful tools for extracting the relevant portions of the circuit under investigation.

See help select for a complete list of actions available in selections.

E. Storing and recalling selections

The current selection can be stored in memory with the command select -set <name>. It can later be recalled using select @<name>. In fact, the @<name> expression pushes the stored selection on the stack maintained by the select command. So for example

```
select @foo @bar %i
```

will select the intersection between the stored selections foo and bar.

In larger investigation efforts it is highly recommended to maintain a script that sets up relevant selections, so they can easily be recalled, for example when Yosys needs to be re-run after a design or source code change.

The history command can be used to list all recent interactive commands. This feature can be useful for creating such a script from the commands used in an interactive session.

V. ADVANCED INVESTIGATION TECHNIQUES

When working with very large modules, it is often not enough to just select the interesting part of the module. Instead it can be useful to extract the interesting part of the circuit into a separate module. This can for example be useful if one wants to run a series of synthesis commands on the critical part of the module and wants to carefully read all the debug output created by the commands in order to spot a problem. This kind of troubleshooting is much easier if the circuit under investigation is encapsulated in a separate module.

Fig. 17 shows how the `submod` command can be used to split the circuit from Fig. 14 and 15 into its components. The `-name` option is used to specify the name of the new module and also the name of the new cell in the current module.

A. Evaluation of combinatorial circuits

The `eval` command can be used to evaluate combinatorial circuits. For example (see Fig. 17 for the circuit diagram of `selstage`):

```
yosys [selstage]> eval -set s2,s1 4'b1001 -set d 4'hc -show n2

9. Executing EVAL pass (evaluate the circuit given an input).
Full command line: eval -set s2,s1 4'b1001 -set d 4'hc -show n2
 Eval result: \n2 = 2'10.
 Eval result: \n1 = 2'10.
```

So the `-set` option is used to set input values and the `-show` option is used to specify the nets to evaluate. If no `-show` option is specified, all selected output ports are used per default.

If a necessary input value is not given, an error is produced. The option `-set-undef` can be used to instead set all unspecified input nets to undef (x).

The `-table` option can be used to create a truth table. For example:

```
yosys [selstage]> eval -set-undef -set d[3:1] 0 -table s1,d[0]

10. Executing EVAL pass (evaluate the circuit given an input).
Full command line: eval -set-undef -set d[3:1] 0 -table s1,d[0]

  \s1 \d [0] |   \n1   \n2
  ---- ------ | ---- ----
  2'00    1'0 | 2'00 2'00
  2'00    1'1 | 2'xx 2'00
  2'01    1'0 | 2'00 2'00
  2'01    1'1 | 2'xx 2'01
  2'10    1'0 | 2'00 2'00
  2'10    1'1 | 2'xx 2'10
  2'11    1'0 | 2'00 2'00
  2'11    1'1 | 2'xx 2'11

 Assumed undef (x) value for the following signals: \s2
```

Note that the `eval` command (as well as the `sat` command discussed in the next sections) does only operate on flattened modules. It can not analyze signals that are passed through design hierarchy levels. So the `flatten` command must be used on modules that instantiate other modules before this commands can be applied.

B. Solving combinatorial SAT problems

Often the opposite of the `eval` command is needed, i.e. the circuits output is given and we want to find the matching input signals. For small circuits with only a few input bits this can be accomplished by trying all possible input combinations, as it is done by the `eval -table` command. For larger circuits however, Yosys provides the `sat` command that uses a SAT [4] solver [5] to solve this kind of problems.
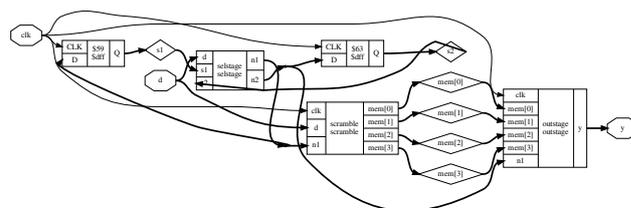
memdemo
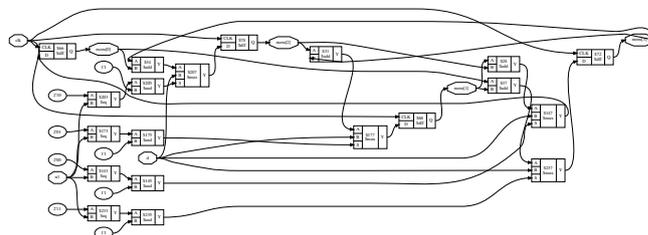
scramble
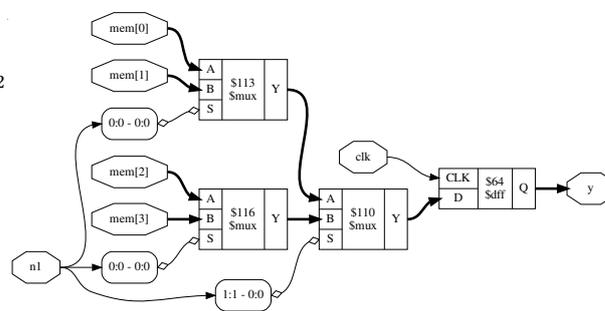
outstage
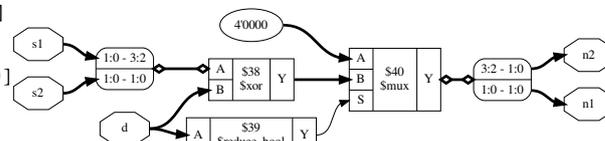
selstage

```
1 | select -set outstage y %ci2:+$dff[Q,D] %ci*:-$mux[S]:-$dff
2 | select -set selstage y %ci2:+$dff[Q,D] %ci*:-$dff @outstage %d
3 | select -set scramble mem* %ci2 %ci*:-$dff mem* %d @selstage %d
4 | submod -name scramble @scramble
5 | submod -name outstage @outstage
6 | submod -name selstage @selstage
```

Figure 17. The circuit from Fig. 14 and 15 broken up using `submod`

The `sat` command works very similar to the `eval` command. The main difference is that it is now also possible to set output values and find the corresponding input values. For Example:

```
yosys [selstage]> sat -show s1,s2,d -set s1 s2 -set n2,n1 4'b1001

11. Executing SAT pass (solving SAT problems in the circuit).
Full command line: sat -show s1,s2,d -set s1 s2 -set n2,n1 4'b1001

 Setting up SAT problem:
 Import set-constraint: \s1 = \s2
 Import set-constraint: { \n2 \n1 } = 4'1001
Final constraint equation: { \n2 \n1 \s1 } = { 4'1001 \s2 }
 Imported 3 cells to SAT database.
 Import show expression: { \s1 \s2 \d }
```

```
Solving problem with 81 variables and 207 clauses..
SAT solving finished - model found:
```

Signal Name	Dec	Hex	Bin
\d	9	9	1001
\s1	0	0	00
\s2	0	0	00

Note that the `sat` command supports signal names in both arguments to the `-set` option. In the above example we used `-set s1 s2` to constraint `s1` and `s2` to be equal. When more complex constraints are needed, a wrapper circuit must be constructed that checks the constraints and signals if the constraint was met using an extra output port, which then can be forced to a value using the `-set` option. (Such a circuit that contains the circuit under test plus additional constraint checking circuitry is called a *miter* circuit.)

Fig. 18 shows a miter circuit that is supposed to be used as a prime number test. If `ok` is 1 for all input values `a` and `b` for a given `p`, then `p` is prime, or at least that is the idea.

The Yosys shell session shown in Fig. 19 demonstrates that SAT solvers can even find the unexpected solutions to a problem: Using integer overflow there actually is a way of "factorizing" 31. The clean solution would of course be to perform the test in 32 bits, for example by replacing `p != a*b` in the miter with `p != {16'd0,a}*b`, or by using a temporary variable for the 32 bit product `a*b`. But as 31 fits well into 8 bits (and as the purpose of this document is to show off Yosys features) we can also simply force the upper 8 bits of `a` and `b` to zero for the `sat` call, as is done in the second command in Fig. 19 (line 31).

The `-prove` option used in this example works similar to `-set`, but tries to find a case in which the two arguments are not equal. If such a case is not found, the property is proven to hold for all inputs that satisfy the other constraints.

It might be worth noting, that SAT solvers are not particularly efficient at factorizing large numbers. But if a small factorization problem occurs as part of a larger circuit problem, the Yosys SAT solver is perfectly capable of solving it.

C. Solving sequential SAT problems

The SAT solver functionality in Yosys can not only be used to solve combinatorial problems, but can also solve sequential problems. Let's consider the entire `memdemo` module from Fig. 14 and suppose we want to know which sequence of input values for `d` will cause the output `y` to produce the sequence 1, 2, 3 from any initial state. Fig. 20 show the solution to this question, as produced by the following command:

```
sat -seq 6 -show y -show d -set-init-undef \
  -max_undef -set-at 4 y 1 -set-at 5 y 2 -set-at 6 y 3
```

The `-seq 6` option instructs the `sat` command to solve a sequential problem in 6 time steps. (Experiments with lower number of steps have show that at least 3 cycles are necessary to bring the circuit in a state from which the sequence 1, 2, 3 can be produced.)

The `-set-init-undef` option tells the `sat` command to initialize all registers to the undef (x) state. The way the x state is treated in Verilog will ensure that the solution will work for any initial state.

The `-max_undef` option instructs the `sat` command to find a solution with a maximum number of undefs. This way we can see clearly which inputs bits are relevant to the solution.

Finally the three `-set-at` options add constraints for the y signal to play the 1, 2, 3 sequence, starting with time step 4.

It is not surprising that the solution sets `d = 0` in the first step, as this is the only way of setting the `s1` and `s2` registers to a known value. The input values for the other steps are a bit harder to work out manually, but the SAT solver finds the correct solution in an instant.

There is much more to write about the `sat` command. For example, there is a set of options that can be used to performs sequential proofs using temporal induction [6]. The command `help sat` can be used to print a list of all options with short descriptions of their functions.

VI. CONCLUSION

Yosys provides a wide range of functions to analyze and investigate designs. For many cases it is sufficient to simply display circuit diagrams, maybe use some additional commands to narrow the scope of the circuit diagrams to the interesting parts of the circuit. But some cases require more than that. For this applications Yosys provides commands that can be used to further inspect the behavior of the circuit, either by evaluating which output values are generated from certain input values (`eval`) or by evaluation which input values and initial conditions can result in a certain behavior at the outputs (`sat`). The SAT command can even be used to prove (or disprove) theorems regarding the circuit, in more advanced cases with the additional help of a miter circuit.

This features can be powerful tools for the circuit designer using Yosys as a utility for building circuits and the software developer using Yosys as a framework for new algorithms alike.

REFERENCES

[1] Clifford Wolf. The Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/
[2] Graphviz - Graph Visualization Software. http://www.graphviz.org/
[3] xdot.py - an interactive viewer for graphs written in Graphviz's dot language. https://github.com/jrfonseca/xdot.py
[4] *Circuit satisfiability problem* on Wikipedia http://en.wikipedia.org/wiki/Circuit_satisfiability
[5] MiniSat: a minimalistic open-source SAT solver. http://minisat.se/
[6] Niklas Een and Niklas Sörensson (2003). Temporal Induction by Incremental SAT Solving. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.4.8161

```
1  module primetest(p, a, b, ok);
2  input [15:0] p, a, b;
3  output ok = p != a*b || a == 1 || b == 1;
4  endmodule
```

Figure 18. A simple miter circuit for testing if a number is prime. But it has a problem (see main text and Fig. 19).

```
 1 ║yosys [primetest]> sat -prove ok 1 -set p 31
 2 ║
 3 ║8. Executing SAT pass (solving SAT problems in the circuit).
 4 ║Full command line: sat -prove ok 1 -set p 31
 5 ║
 6 ║Setting up SAT problem:
 7 ║Import set-constraint: \p = 16'0000000000011111
 8 ║Final constraint equation: \p = 16'0000000000011111
 9 ║Imported 6 cells to SAT database.
10 ║Import proof-constraint: \ok = 1'1
11 ║Final proof equation: \ok = 1'1
12 ║
13 ║Solving problem with 2790 variables and 8241 clauses..
14 ║SAT proof finished - model found: FAIL!
15 ║
16 ║   _____                 ___        ___       _ _              _ _
17 ║  (_____  \               / __)      / __)     (_) |            | | |
18 ║   _____) )___ ___ ___ _| |__   _| |__ _____ _| | _____ __| | | |
19 ║  |  ____/ ___) _ \ / _ (_   __)  (_   __|____ | | || ___ |/ _ |_|
20 ║  | |    | | | |_| | |_| | | |       | | / ___ | | || ___( (_| |_
21 ║  |_|    |_|  \___/ \___/ |_|       |_|  \_____|_|\_)_____)\____|_|
22 ║
23 ║
24 ║   Signal Name              Dec       Hex                   Bin
25 ║  ------------------- ---------- ---------- --------------------
26 ║   \a                     15029      3ab5      0011101010110101
27 ║   \b                      4099      1003      0001000000000011
28 ║   \ok                        0         0                     0
29 ║   \p                        31        1f      0000000000011111
30 ║
31 ║yosys [primetest]> sat -prove ok 1 -set p 31 -set a[15:8],b[15:8] 0
32 ║
33 ║9. Executing SAT pass (solving SAT problems in the circuit).
34 ║Full command line: sat -prove ok 1 -set p 31 -set a[15:8],b[15:8] 0
35 ║
36 ║Setting up SAT problem:
37 ║Import set-constraint: \p = 16'0000000000011111
38 ║Import set-constraint: { \a [15:8] \b [15:8] } = 16'0000000000000000
39 ║Final constraint equation: { \a [15:8] \b [15:8] \p } = { 16'0000000000000000 16'0000000000011111 }
40 ║Imported 6 cells to SAT database.
41 ║Import proof-constraint: \ok = 1'1
42 ║Final proof equation: \ok = 1'1
43 ║
44 ║Solving problem with 2790 variables and 8257 clauses..
45 ║SAT proof finished - no model found: SUCCESS!
46 ║
47 ║                  /$$$$$$      /$$$$$$$$     /$$$$$$$
48 ║                 /$$__  $$    | $$_____/    | $$__  $$
49 ║                | $$  \ $$    | $$          | $$  \ $$
50 ║                | $$  | $$    | $$$$$       | $$  | $$
51 ║                | $$  | $$    | $$__/       | $$  | $$
52 ║                | $$/$$ $$    | $$          | $$  | $$
53 ║                |  $$$$$$/ /$$| $$$$$$$$ /$$| $$$$$$$//$$
54 ║                 \____ $$$|__/|_____/|__/|_____/|__/
55 ║                      \__/
```

Figure 19. Experiments with the miter circuit from Fig. 18. The first attempt of proving that 31 is prime failed because the SAT solver found a creative way of factorizing 31 using integer overflow.

```
1  yosys [memdemo]> sat -seq 6 -show y -show d -set-init-undef \
2             -max_undef -set-at 4 y 1 -set-at 5 y 2 -set-at 6 y 3
3
4  6. Executing SAT pass (solving SAT problems in the circuit).
5  Full command line: sat -seq 6 -show y -show d -set-init-undef
6             -max_undef -set-at 4 y 1 -set-at 5 y 2 -set-at 6 y 3
7
8  Setting up time step 1:
9  Final constraint equation: { } = { }
10 Imported 29 cells to SAT database.
11
12 Setting up time step 2:
13 Final constraint equation: { } = { }
14 Imported 29 cells to SAT database.
15
16 Setting up time step 3:
17 Final constraint equation: { } = { }
18 Imported 29 cells to SAT database.
19
20 Setting up time step 4:
21 Import set-constraint for timestep: \y = 4'0001
22 Final constraint equation: \y = 4'0001
23 Imported 29 cells to SAT database.
24
25 Setting up time step 5:
26 Import set-constraint for timestep: \y = 4'0010
27 Final constraint equation: \y = 4'0010
28 Imported 29 cells to SAT database.
29
30 Setting up time step 6:
31 Import set-constraint for timestep: \y = 4'0011
32 Final constraint equation: \y = 4'0011
33 Imported 29 cells to SAT database.
34
35 Setting up initial state:
36 Final constraint equation: { \y \s2 \s1 \mem[3] \mem[2] \mem[1]
37                             \mem[0] } = 24'xxxxxxxxxxxxxxxxxxxxxxxx
38
39 Import show expression: \y
40 Import show expression: \d
41
42 Solving problem with 10322 variables and 27881 clauses..
43 SAT model found. maximizing number of undefs.
44 SAT solving finished - model found:
45
46   Time Signal Name            Dec
   Hex||         Bin                                        |
47 || ---- ------------------- ---------- ---------- ----+----------
48 ||  init \mem[0]               --              |
   -- ||         xxxx                                       |
49 ||  init \mem[1]               --              |
   -- ||         xxxx                                       |
50 ||  init \mem[2]               --              |
   -- ||         xxxx                                       |
51 ||  init \mem[3]               --              |
   -- ||         xxxx                                       |
52 ||  init \s1                   --              |
   -- ||         xx                                         |
53 ||  init \s2                   --              |
   -- ||         xx                                         |
54 ||  init \y                    --              |
   -- ||         xxxx                                       |
55 || ---- ------------------- ---------- ---------- ----+----------
56 ||     1 \d                     0               |
   0  ||         0000                                       |
57 ||     1 \y                    --              |
   -- ||         xxxx                                       |
58 || ---- ------------------- ---------- ---------- ----+----------
59 ||     2 \d                     1               |
   1  ||         0001                                       |
60 ||     2 \y                    --              |
   -- ||         xxxx                                       |
61 || ---- ------------------- ---------- ---------- ----+----------
62 ||     3 \d                     2               |
   2  ||         0010                                       |
63 ||     3 \y                     0               |
   0  ||         0000                                       |
64 || ---- ------------------- ---------- ---------- ----+----------
65 ||     4 \d                     3               |
   3  ||         0011                                       |
66 ||     4 \y                     1               |
   1  ||         0001                                       |
67 || ---- ------------------- ---------- ---------- ----+----------
68 ||     5 \d                    --              |
   -- ||         001x                                       |
69 ||     5 \y                     2               |
   2  ||         0010                                       |
70 || ---- ------------------- ---------- ---------- ----+----------
71 ||     6 \d                    --              |
   -- ||         xxxx                                       |
72 ||     6 \y                     3               |
   3  ||         0011                                       |
```

Figure 20. Solving a sequential SAT problem in the memdemo module from
Fig. 14.