Note: Forum thread which lead to the below is found at:

https://sourceforge.net/p/ngspice/discussion/ngspice-tips/thread/6a5b9dd2/

1. Install latest version of verilator:

To run this, verilator version 4.2.10 or later is required. Chances are the version which comes bundeled with your linux distro will be older than this. You can check with the below link:

https://repology.org/project/verilator/versions

For example, this link shows version 4.038 comes with Ubuntu 22.04.

To update the latest version follow the instructions on the below link:

https://verilator.org/guide/latest/install.html

Specifically this means doing the following:

```
git clone https://github.com/verilator/verilator
unset VERILATOR_ROOT
cd verilator
autoconf
./configure
make
sudo make install
```

2. Run the example code created by Giles.

This can be found at:

https://ngspice.sourceforge.io/docs/others/Verilog-CoSim.pdf

To do this, download the files at the following links:

https://sourceforge.net/p/ngspice/ngspice/ci/master/tree/src/xspice/verilog/
https://sourceforge.net/p/ngspice/ngspice/ci/master/tree/examples/xspice/verilator

Download to a directory naming structure like: <name_top>/<name>

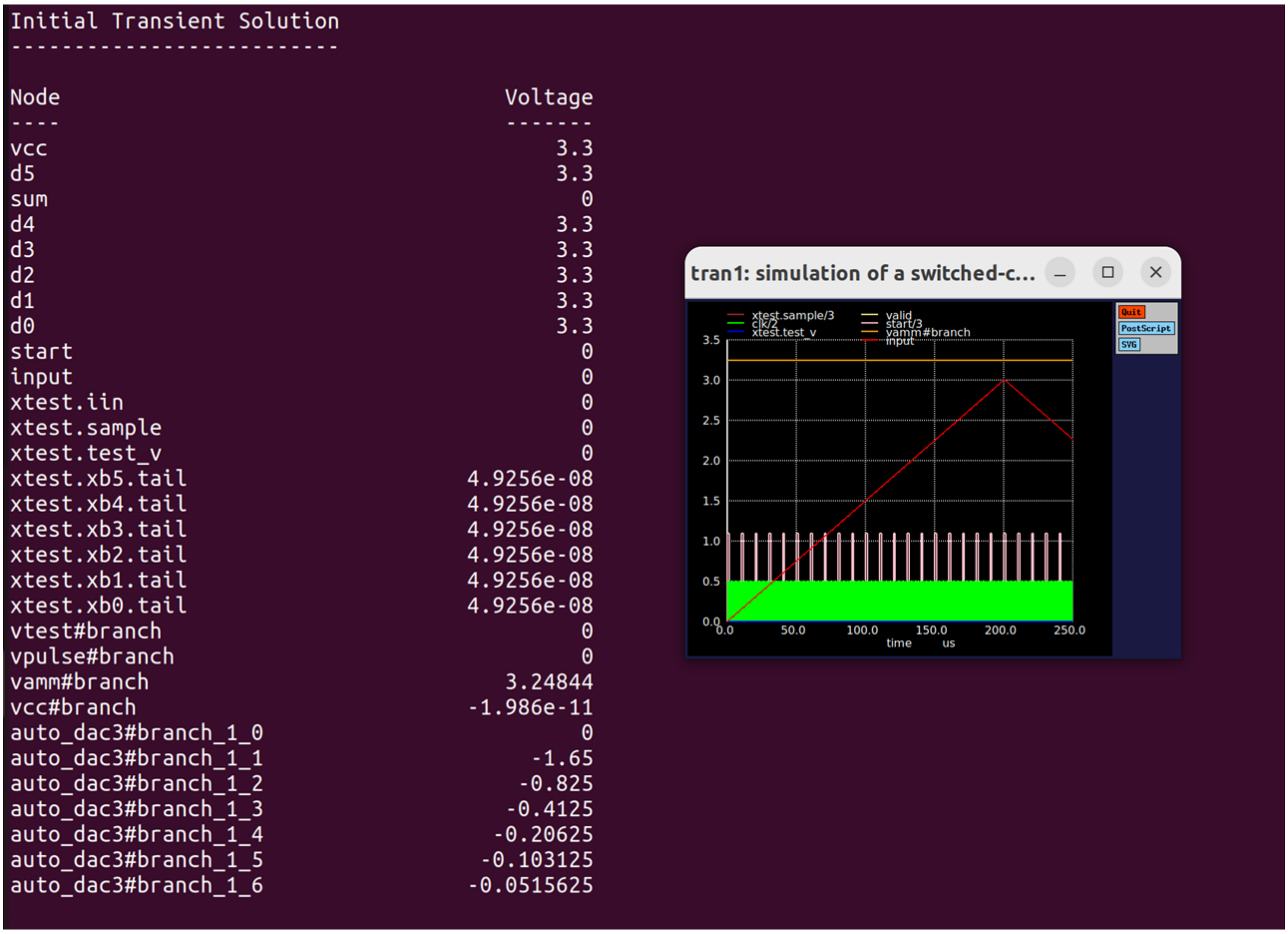
The idea here is that you will store the files in the <name> directory and move one level above (to <name_top> to run the following command:

```
ngspice <name>/vlnggen adc.v
```

After this, go into <name> and run:

```
ngspice adc.cir
```

You should get an output like the below:



```
Initial Transient Solution
--------------------------

Node                                   Voltage
----                                   -------
vcc                                        3.3
d5                                         3.3
sum                                          0
d4                                         3.3
d3                                         3.3
d2                                         3.3
d1                                         3.3
d0                                         3.3
start                                        0
input                                        0
xtest.iin                                    0
xtest.sample                                 0
xtest.test_v                                 0
xtest.xb5.tail                     4.9256e-08
xtest.xb4.tail                     4.9256e-08
xtest.xb3.tail                     4.9256e-08
xtest.xb2.tail                     4.9256e-08
xtest.xb1.tail                     4.9256e-08
xtest.xb0.tail                     4.9256e-08
vtest#branch                                 0
vpulse#branch                                0
vamm#branch                            3.24844
vcc#branch                          -1.986e-11
auto_dac3#branch_1_0                         0
auto_dac3#branch_1_1                     -1.65
auto_dac3#branch_1_2                    -0.825
auto_dac3#branch_1_3                   -0.4125
auto_dac3#branch_1_4                  -0.20625
auto_dac3#branch_1_5                 -0.103125
auto_dac3#branch_1_6                -0.0515625
```

3. Create an ‘n’ bit counter in verilog:

In the below example I have created a 4bit counter:

****

```
module counter(out, clk, reset);

  parameter WIDTH = 4;

  output [WIDTH-1 : 0] out;
  input             clk, reset;

  reg [WIDTH-1 : 0]   out;
  wire              clk, reset;

  always @(posedge clk or posedge reset)
    if (reset)
      out <= 0;
    else
      out <= out + 1;

endmodule // counter
```

In the above example the always loop is triggered upon a +ve clock or reset edge. Once done, if reset is high, clock outputs a 0. Otherwise it increments the output upon each +ve clock edge. This means that until reset goes hi, the output is undefined.

4. Create a tb to test that counter in verilog:

In the below example, I create a tb which asserts reset after 10 clocks and de-aserts it after 5 clocks, stopping after 200 clocks. The period of the clock sent to the counter is then 10 clocks.

****

```
module counter_tb;

  /* Make a reset that pulses once. */
  reg reset = 0;
  initial begin
     $dumpfile("counter_data.vcd");
     $dumpvars(0,counter_tb);
     # 10 reset = 1;
     # 5 reset = 0;
     # 200 $stop;
  end

  /* Make a regular pulsing clock. */
  reg clk = 0;
  always #5 clk = !clk;

  wire [3:0] value;
  counter c1 (value, clk, reset);

  initial
     $monitor("At time %t, value = %h (%0d)",
            $time, value, value);
endmodule // test
```

****

What we will expect when we run this is that after 10 clocks reset will go high. Its +ve edge will activate the always loop in the verilog file which will set the output to 0. This will only last half a clock period (after which point reset goes low). Then clock will increment from there.

5. Install icarus:

Icarus will be used to verify the verilog code. Its full installation instructions are found at the below link:

https://steveicarus.github.io/iverilog/usage/installation.html

6. Install gtk:

Icarus is used to compile and run verilog code but needs gtk to view the outputs. My experience with the gtk install was not good when I attempted a manual install. After a few hours of unsuccessfully trying this I resorted to installing the version which came with Ubuntu (using sudo agt get). Gtk is a very mature digital waveviewer so the version with Ubuntu (altho prob not the most up to date version) worked fine.

7. Run verification:

As described at the following link:

https://steveicarus.github.io/iverilog/usage/gtkwave.html

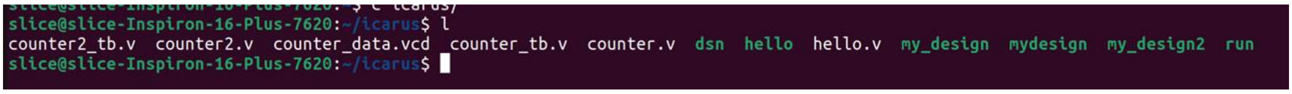
Compile the verilog files created in steps 3 and 4 using icarus with the following code:

```
iverilog -o dsn counter_tb.v counter.v
```

This compiles into a file called dsn. Run this using the below:

```
vvp dsn
```

This dumps all the data to a .vcd file called "counter_data.vcd" as per below:

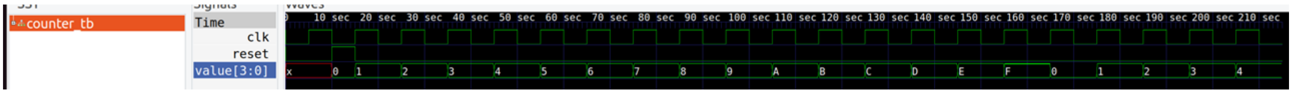


```
slice@slice-Inspiron-16-Plus-7620:~/icarus$ l
counter2_tb.v  counter2.v  counter_data.vcd  counter_tb.v  counter.v  dsn  hello  hello.v  my_design  mydesign  my_design2  run
slice@slice-Inspiron-16-Plus-7620:~/icarus$ 
```

gtkwave then needs to be used to view this .vcd file with the below command:

```
gtkwave counter_data.vcd &
```

The results is shown below:



Behaviour is as expected. Counter output is undefined until reset goes high whose +ve edge sends it low for half a clock period. Once low, the next clock increments the count to 16 after which of course it resets.

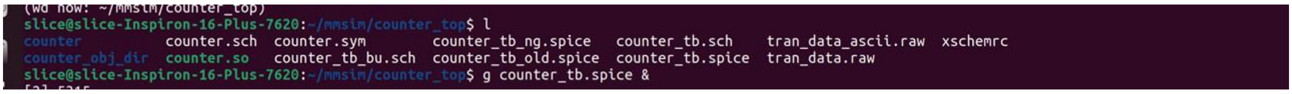
So in a nutshell, commands to compile/run/verify are:

```
# compile vlog files
iverilog -o dsn counter_tb.v counter.v

# execute vlog files
vvp dsn

#open gtkwave to view them
gtkwave counter_data.vcd &
```

To be efficient you can place these in an executable (called run) using chmod + x. Now by typing /.run you will do all the above.

8. Convert .v to .so file:

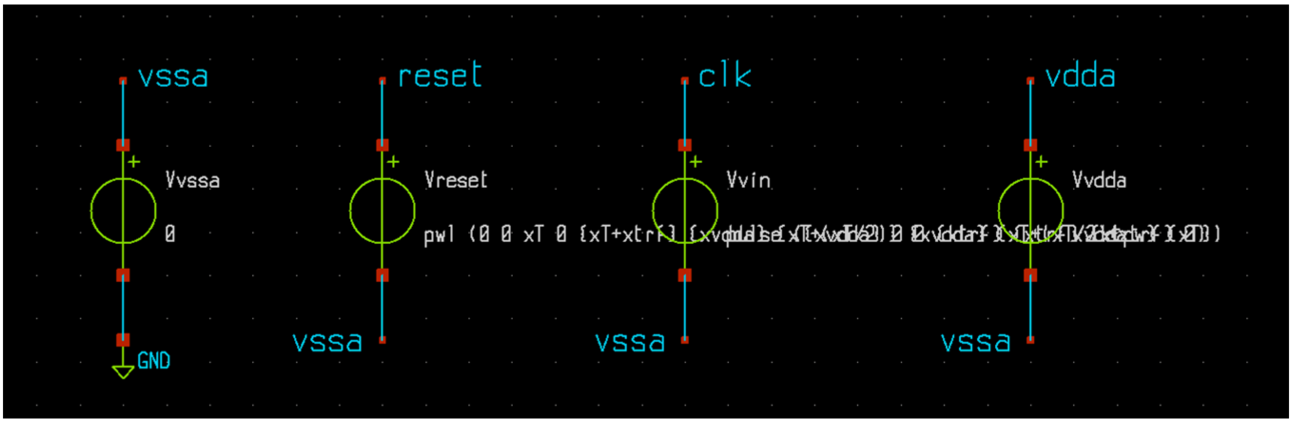
Create a directory structure like /counter_top/counter.
In directory counter, place the verilog code created in step 3.
Go one level above and write the line: ngspice counter/vlnggen counter.v
This will create all the necessary files, and the all important .so file which is what will be used to represent the counter in verilog:



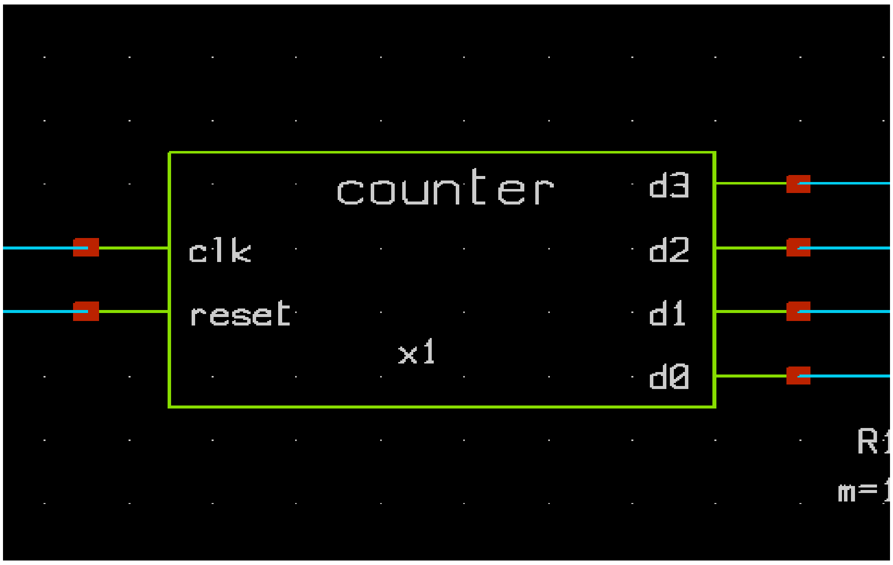
```
(wd now: ~/mmsim/counter_top)
slice@slice-Inspiron-16-Plus-7620:~/mmsim/counter_top$ l
counter          counter.sch  counter.sym         counter_tb_ng.spice   counter_tb.sch    tran_data_ascii.raw  xschemrc
counter_obj_dir  counter.so   counter_tb_bu.sch   counter_tb_old.spice  counter_tb.spice  tran_data.raw
slice@slice-Inspiron-16-Plus-7620:~/mmsim/counter_top$ g counter_tb.spice &
```

9. Create a tb in xschem:

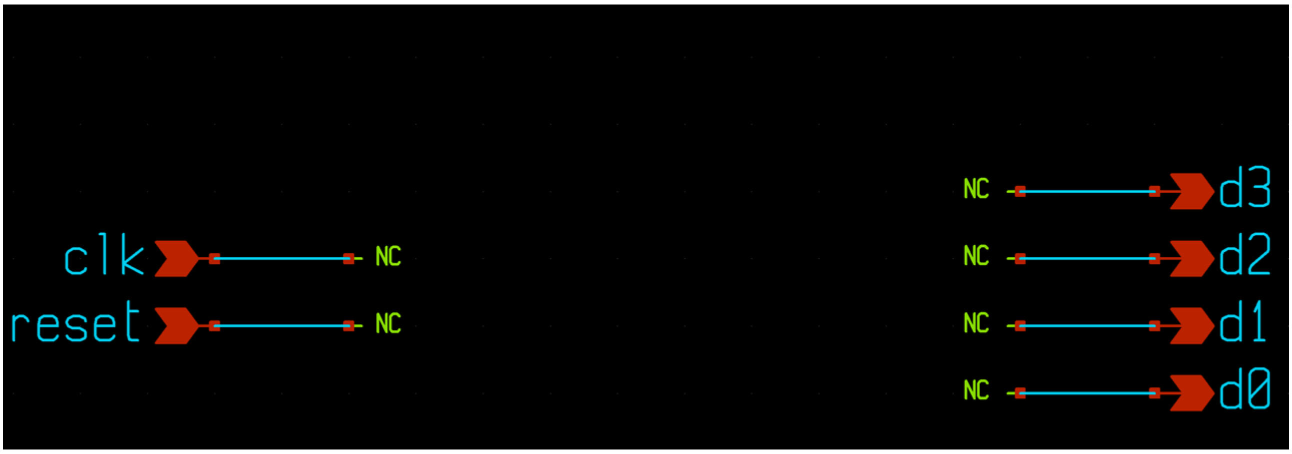
You will be running all this through xschem so use that to create the required stimulus. As shown below this is a ground, reset, clk and vdda (stimulus details are shown later).



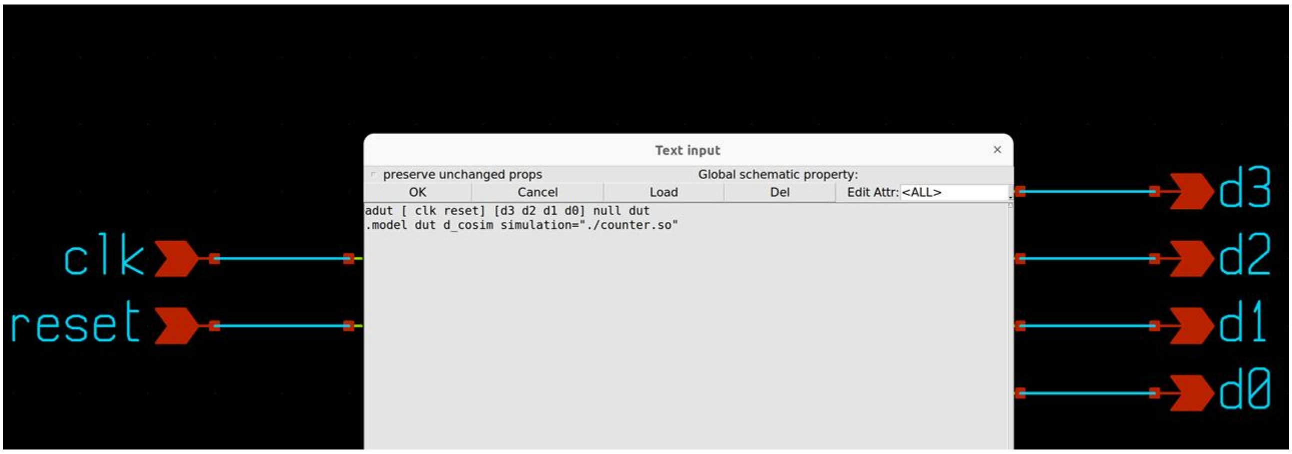
Next create a counter symbol with all the required pins:



Inside that symbol, place noConns on all the pins (otherwise xschem will error out):



Next ‘q’ on the symbol and place the following in the text field:



```
adut [ clk reset] [d3 d2 d1 d0] null dut
.model dut d_cosim simulation="./counter.so"
```

This code is pretty generic and to get a true understanding of it, the user is directed to the following sections of the ngspice manual:

Relevant sections are 12.1 XSPICE (device and model card format), 14.3 (digital devices with behaviour defined by Verilog) and 12.4.25 (the d_cosim code model that is used to create such simulated devices). Assuming there are analogue nodes and devices in the circuit, 12.6 (automatic analogue-digital bridges) is also relevant.

Looking at the above one can see the adut command format as [inputs] [outputs]. In addition, the above assumes the .so file is saved in the same directory within which xschem is running.

Netlisting will then give you the below:

```
*****************************************************************************

** sch_path: /home/slice/mmsim/counter_top/counter_tb.sch
**.subckt counter_tb
Vvssa vssa GND 0
```

Vvin clk vssa pulse ({xvdda} 0 0 {xtrf} {xtrf} {xtpw} {xT})
Vreset reset vssa pwl (0 0 xT 0 {xT+xtrf} {xvdda} {xT+(xT/2)} {xvdda} {xT+(xT/2)+xtrf} 0)
x1 d3 clk d2 reset d1 d0 counter

```
R1 d0 vssa 1k m=1
R2 d1 vssa 1k m=1
R3 d2 vssa 1k m=1
R4 d3 vssa 1k m=1
Vvdda vdda vssa xvdda
**** begin user architecture code

* Parameters
.param xvdda = 3.3
.param xtrf = 50p
.param xfreq = 1Meg
.param xT = {1/xfreq}
.param xtpw = {(xT/2)-xtrf}
.param numPeriods = 17
.csparam xtsim = {numPeriods*xT}

  .control

  save all
  echo $&xtsim

** 1. TRAN SIM **

  tran 1n $&xtsim

  remzerovec
  write tran_data.raw

  *plot tran.v(clk) tran.v(reset)
  *plot tran.v(reset)
  *plot clk reset+2 d0+4 d1+5.5 d2+7 d3+8.5

  *compose d0 xspice

  set filetype=ascii
  remzerovec
  write tran_data_ascii.raw

  setplot

  .endc

**** end user architecture code
**.ends

* expanding   symbol:  counter.sym # of pins=6
** sym_path: /home/slice/mmsim/counter_top/counter.sym
** sch_path: /home/slice/mmsim/counter_top/counter

.subckt counter d3 clk d2 reset d1 d0
*.ipin clk
*.ipin reset
```

```
*.opin d3
*.opin d2
*.opin d1
*.opin d0
* noconn d3
* noconn d2
* noconn d1
* noconn d0
* noconn clk
* noconn reset
**** begin user architecture code
adut [ clk reset] [d3 d2 d1 d0] null dut
.model dut d_cosim simulation="./counter.so"
**** end user architecture code
.ends

.GLOBAL GND
.end
```

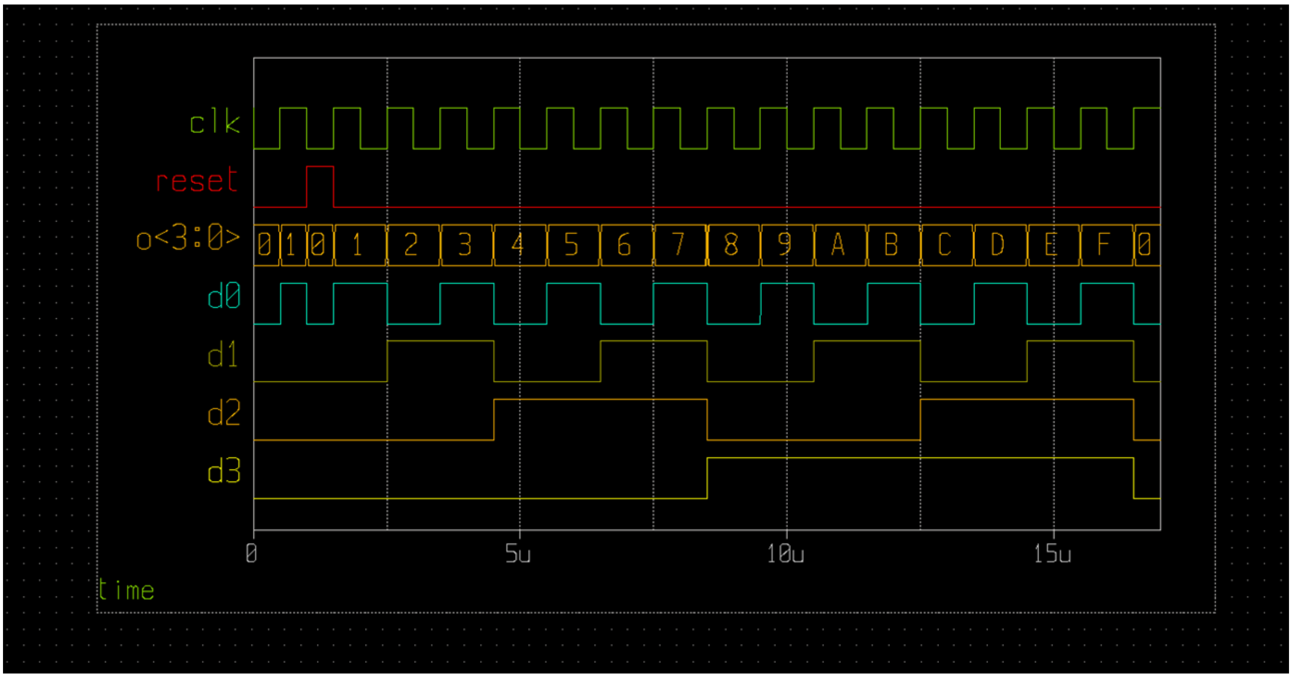
****

In the above I have highlighted the nets which refer to the counter and its subckt definition. In addition I have made in bold the stimulus as it shows how to pass variables to them, as per the below thread:

https://sourceforge.net/p/ngspice/discussion/127605/thread/2d238b9f6c/

10. View the outputs using xschem waveviewer:

After a netlist and run, you can view the results in xschem as per the below:



Cross checking with the gtk results in step 7 should match perfectly.

11. Example:

A good eample is located at:

~/mmsim/counter_top

This will produce all the plots shown throughout this doc.

This example can also be found on the below public repository in github:

https://github.com/SLICESemiconductor/OpenSourceTool_Examples/tree/main/Running_vlog_in_ngspice

12. Conclusion:

This shows the basic setup of creating a verilog file, verifying in a digital simulator (icarus) and converting to a form which ngspice will understand before re-running it inside there (through xschem). We can build from this example to make more complex ones but they should all follow this general flow. If and when you have issues, email the forum and Giles should pick up the ticket.

Note: When I finally create a ppt of this I will need to store the code in git to make it publicly available. Will be doing this once I have the Yosys synthesis working and the behavioural view scrpts from Tim in place.