Welcome to this guide for Open Source Mixed Signal Design!

This guide will help you understand the principles of mixed-signal design using open-source tools. We will cover topics such as:

- Digital design with SystemVerilog
- Analog design with SPICE with XSCHEM
- Cosimulation of digital and analog components with ngspice
- Visualization of simulation results with XSCHEM and Python

By the end of this guide, you will have a solid understanding of how to create and simulate mixed-signal designs using open-source tools.

PDK Examples

Exploring the PDK Examples

The PDK (Process Design Kit) examples provide a set of pre-configured designs and simulations that can be used as a starting point for your own projects. These examples cover a wide range of use cases and demonstrate best practices for using the tools and libraries available in the PDK.

To explore them, simply open XSCHEM:

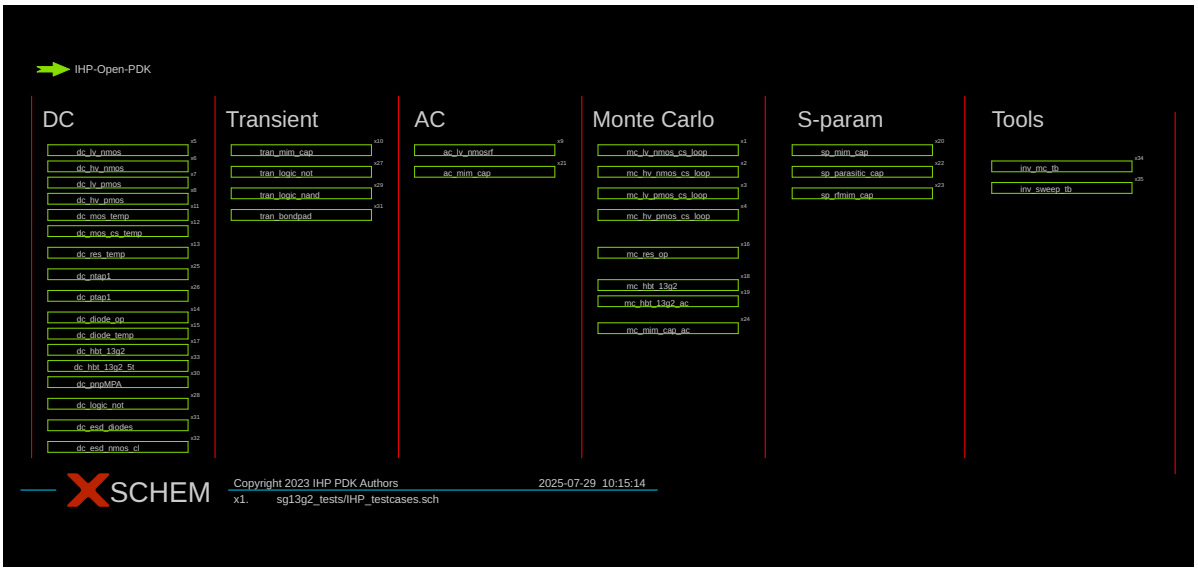
```
xschem
```

You should see this start page:

Here, you can see some useful commands and blocks which you can place into your own designs to the right. On the top right, you can access a cheatsheet of the controls available in XSCHEM.

We are interested in the examples to the left, those for ngspice to be exact. Jump into this sub-sheet by pressing `E` while hovering over the green rectangle (`IHP_testcases`) below `NGSPICE` .

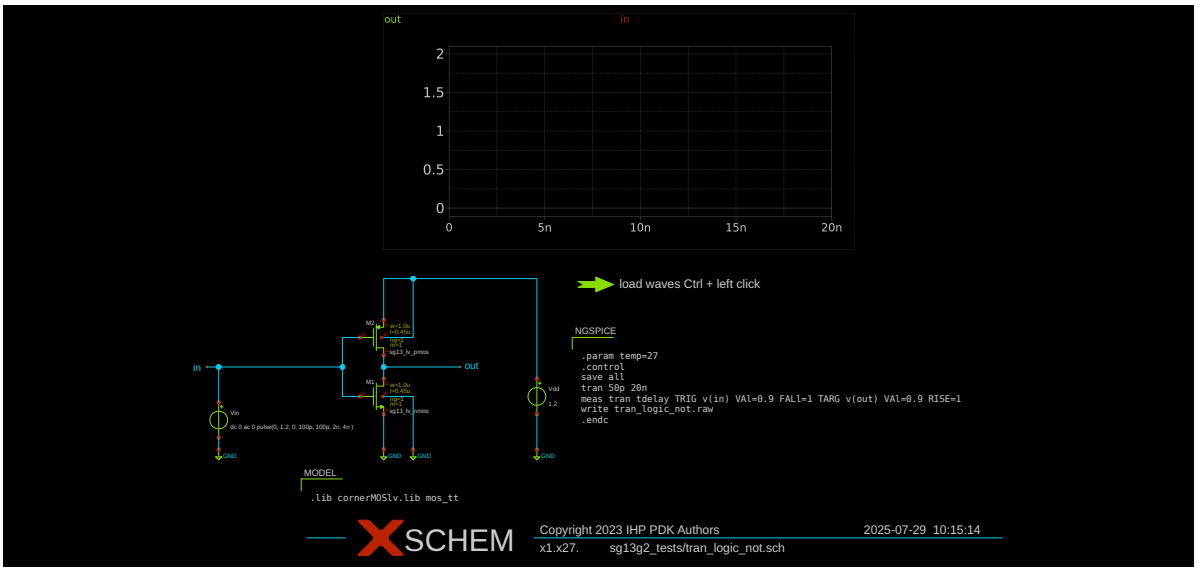
You should see this sheet:



Here, you can explore various test cases and examples for different analysis types. You can once again enter each test case by pressing `E` while hovering over it. Leave a sub-sheet by pressing `Ctrl + E`.

Simulating a Test Case - Inverter

We explore the `tran_logic_not` test case. This simulates a logic inverter using a transient analysis. Please enter it. You should see this sheet:



Let's just simulate the inverter first.

In order to do a simulation, we first need to generate the netlist. The netlist describes the device instances and their connections in the circuit. In XSCHEM, you can generate the netlist by clicking on the "Netlist" button in the toolbar.

After the netlist is generated, you can run the simulation by clicking on the "Simulate" button in the toolbar. This will open a new window, which is the interactive prompt for ngspice.

Simulation Log

You should see this log output:

```
*************
** ngspice-44.2 Circuit level simulation program
** Compiled with KLU Direct Linear Solver
** The U. C. Berkeley CAD Group
** Copyright 1985-1994, Regents of the University of California.
** Copyright 2001-2024, The ngspice team.
** Please get your ngspice manual from https://ngspice.sourceforge.io/docs.html
** Please file your bug-reports at http://ngspice.sourceforge.net/bugrep.html
** Creation Date: Tue Jul 29 08:21:22 UTC 2025
********

Note: No compatibility mode selected!

Circuit: ** sch_path: /foss/pdks/ihp-
sg1392/libs.tech/xschem/s91392_tests/tran_logic_not.sch

Doing analysis at TEMP = 27.000000 and TNOM = 27.000000

Using SPARSE 1.3 as Direct Linear Solver

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

Node                                   Voltage
----                                   -------
in                                           0
net1                                       1.2
out                                        1.2
vdd#branch                          -2.67579e-10
vin#branch                           4.03724e-11

 Reference value: 1.21150e-08
No. of Data Rows : 465
tdelay               = 1.119817e-10 targ= 2.236982e-09 trig= 2.125000e-09
binary raw file "tran_logic_not.raw"
ngspice 1 ->
```

This output shows that the simulation was successful and provides information about the circuit being simulated.

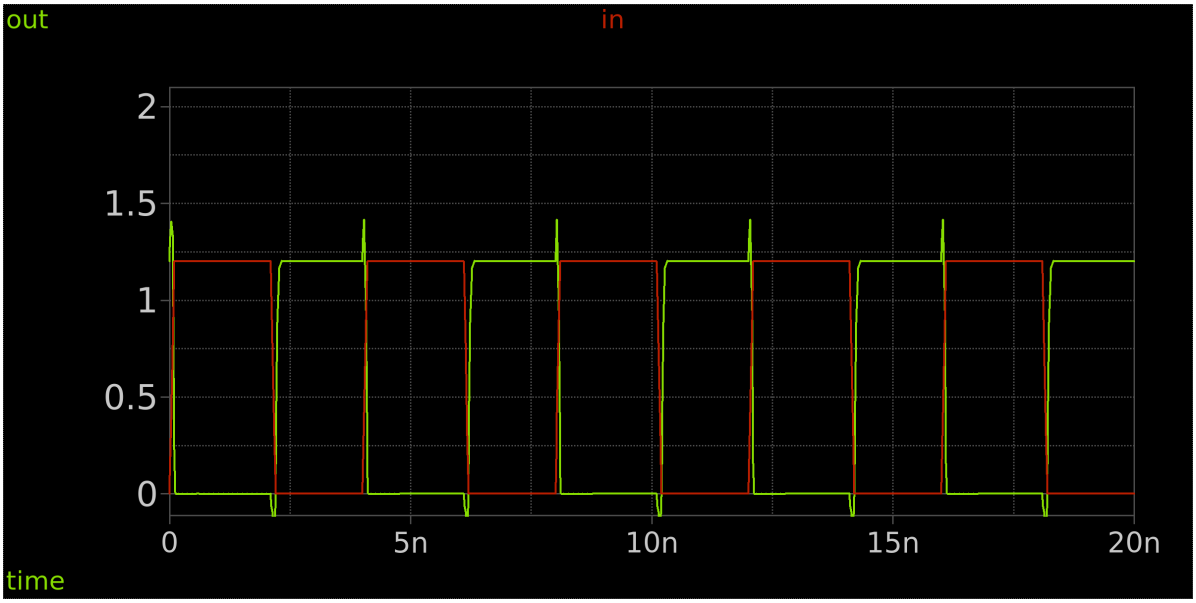
This includes:

- The temperature at which the simulation was run
- The initial transient solution (values at each node at time t=0)
- The amount of data generated during the simulation (`No. of Data Rows`)
- Measurement results (e.g., propagation delay `tdelay` , user-defined by `.meas` commands)
- The location of the binary raw file containing the simulation data (`tran_logic_not.raw`)

Graphs

The simulation results in the resulting raw file can also be plotted. You can do this by using the waveform reload launcher (`load waves Ctrl + left click`).

This should load the traces into the plot in the schematic editor. It should look something like this:



We can see that the inverter is functioning as expected, with the output being the inverse of the input.

Measurements/Analysis

You can already do some analysis in ngspice directly via `.meas` commands. In the case of this simulation, we have this command:

```
meas tran tdelay TRIG v(in) VAl=0.9 FALl=1 TARG v(out) VAl=0.9 RISE=1
```

This measurement command measures the propagation delay (`tdelay`) of the inverter by triggering on the input voltage (`v(in)` at 0.9V falling edge) and targeting the output voltage (`v(out)` at 0.9V rising edge). From the log output, we can see that the measured propagation delay is 0.112 ns, where we triggered at the falling input edge at 2.125 ns and registered the output rising edge at 2.237 ns.

Later, we take a look on how to look at this raw data and perform more in-depth analysis on it in Python.

Modifying component parameters

Staying inside this test case, we now want to see how the behaviour of the inverter changes depending on the transistor sizing. We can do this by modifying the width and length of the transistors in the inverter circuit.

In XSCHEM, you can easily modify component parameters by selecting the component (double click on component) and editing its properties in the property editor. For example, to change the width of a transistor, you can change the `w` parameter.

After making changes to the transistor sizes, you can regenerate the netlist and run the simulation again to see how the changes affect the inverter's performance.

Modifying simulation parameters

To define the behaviour of the simulation, each top-level sheet should include simulation commands. They are placed in a code block like this (double click on it to edit):

```
name=NGSPICE only_toplevel=true
value="
.param temp=27
.control
save all
tran 50p 20n
meas tran tdelay TRIG v(in) VAl=0.9 FALl=1 TARG v(out) VAl=0.9 RISE=1
write tran_logic_not.raw
.endc
"
```

Here, `name` simply specifies the title used for the code block, `only_toplevel` indicates that the commands should only be applied to the top-level sheet, and `value` contains the actual simulation commands to be executed. Let's take a closer look at the commands used in this example.

```
.param temp=27
.control
save all
tran 50p 20n
meas tran tdelay TRIG v(in) VAl=0.9 FALl=1 TARG v(out) VAl=0.9 RISE=1
write tran_logic_not.raw
.endc
```

- The `.param` command sets a simulation parameter (in this case, temperature).
- The `.control` block contains the simulation commands to be executed (**Note:** There are multiple ways to do this. We will just focus on this one for now).
- The `save all` command tells the simulator to save all node voltages and branch currents.
- The `tran 50p 20n` command performs a transient analysis with a maximum time step of 50 ps and a total simulation time of 20 ns.
- The `meas` command measures the propagation delay of the inverter (as described above).
- The `write` command saves the simulation results to a raw file.
- The `.endc` command ends the control (`.control`) block.

Now, if we want to modify the simulation parameters, we can do so by editing the code block accordingly. For example, we could change the temperature or the transient analysis time settings:

```
.param temp=10
.control
save all
tran 50p 10n
meas tran tdelay TRIG v(in) VAl=0.9 FALl=1 TARG v(out) VAl=0.9 RISE=1
write tran_logic_not.raw
.endc
```

Here, we simulate the inverter at 10 degrees Celsius with a total simulation time of 10 ns.

Creating a New Design

Creating a new Block

Here, we will create our own inverter design, including a sub-sheet, symbol and testbench.

Creating the Schematic

To create a new schematic in XSCHEM, we simply create a new `.sch` file (schematic format for XSCHEM). We can do this via

```
xschem inverter.sch
```

You will get a warning that the file does not exist, which is fine.

Now, you should see a blank schematic editor window where you can start building your inverter design.

Symbols

We start by placing our two MOSFETs (NMOS and PMOS) in the schematic. To do this, select the `Insert Symbol` symbol from the toolbar (looks like an AND gate) or press `Insert` on your keyboard.

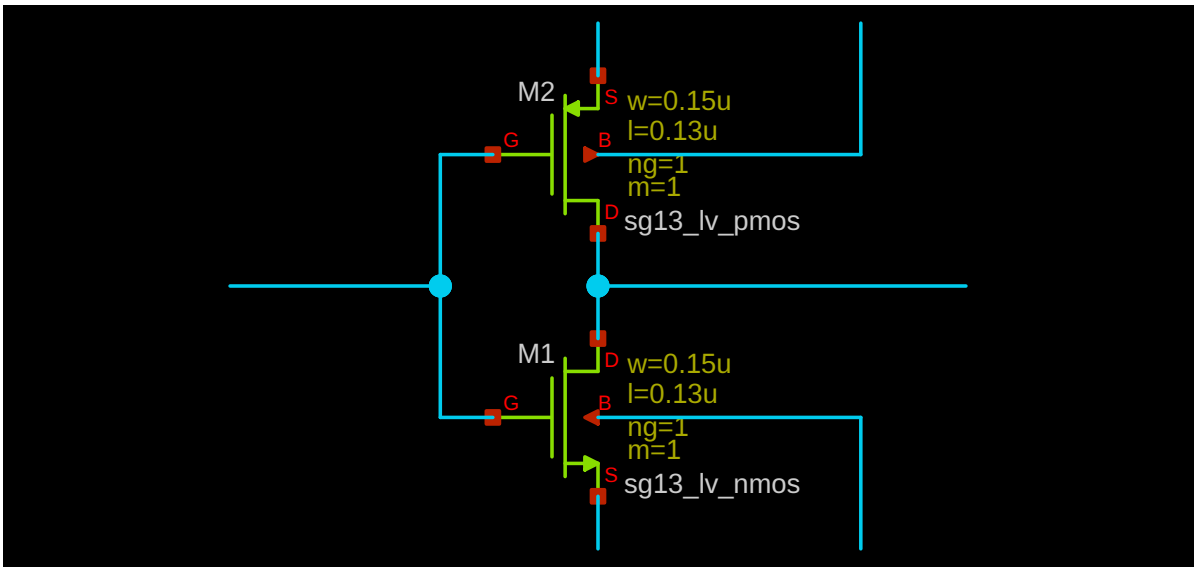
This will open the `Choose symbol` dialog, where we can see some libraries. In our case, we choose the one ending with `ihp-sg13g2/libs.tech/xschem`. There, we choose `sg23g2_pr` and `sg13_lv_nmos`. Click `OK` to insert the symbol into the schematic. Place the symbol by moving it somewhere sensible and by then left-clicking to confirm the placement.

Repeat this for the PMOS transistor by selecting `sg13_lv_pmos` from the same library.

Wires

Now, we connect the gates and the drains of the NMOS and PMOS transistors together to form the inverter structure. You can place a wire by pressing `W` on your keyboard while you are at your first connection point. If you want to have a corner in your wire, move the cursor to the position of the corner and press `W` again. This will place the first wire segment. Then, when you are at the second connection point, simply left-click to finalize the wire placement.

Also place some short wires out of the sources and bodies of the transistors as well as some input and output wires out of the gates/drains. Your design should look something like this:



Labels

Now, let's place some labels to identify each net. To do this, open the `Choose symbol` dialog by pressing `Insert` again and go to the builtin devices library. You can do this by pressing the `Home` button on the bottom left. This will choose the `xschem_library/devices` library. These are the spice primitives (not physical devices).

We want to search for the `lab` symbols. You can do this by clicking into the search field on the bottom left and by entering `lab`. This will give us some different options, but we are taking the `lab_pin.sym` symbol. Select it and place it on the end of the input wire.

To copy a symbol instance, select the label and press `C`. This will attach a new instance to the cursor. You can rotate the current instance by pressing `Ctrl + R`.

Repeat this for the output wire and sources/bodies wires of the transistors.

To rename a label, simply double-click on it and enter the new name (`lab` parameter). Name the labels:

- Input: `in`
- Output: `out`
- NMOS source/body: `VSS`
- PMOS source/body: `VDD`

Pins

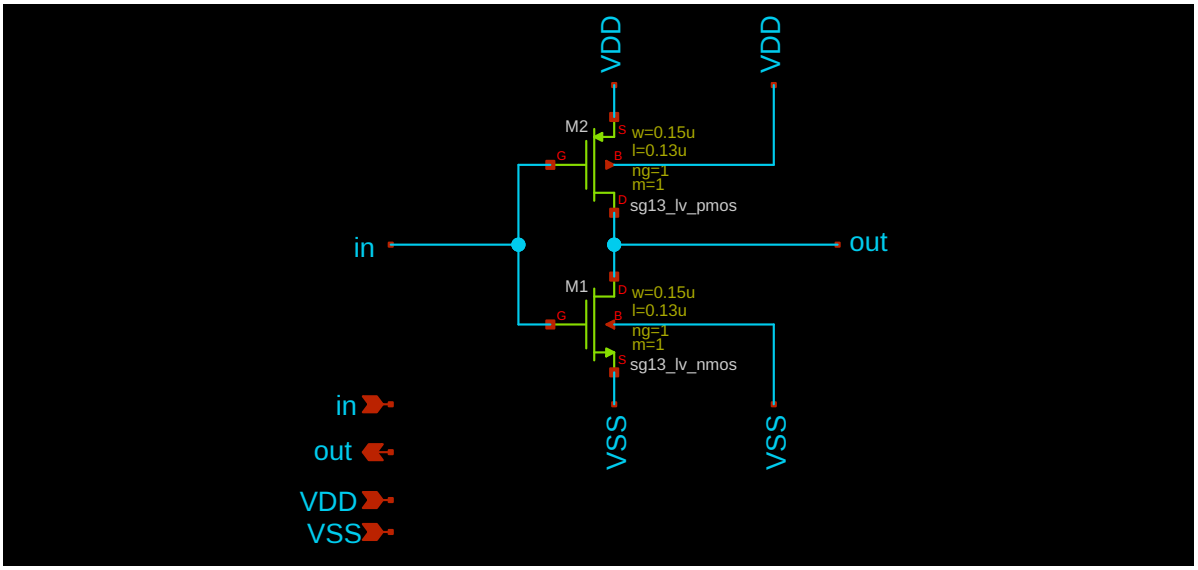
Since we want the inverter to be a symbol in the end, we need to place to place the pins of the symbol. To do this, we use `ipin.sym` for inputs, `opin.sym` for outputs and `iopin.sym` for bidirectional pins from the generic library.

Place the following pins:

- Input: `ipin.sym`, name it `in`
- Output: `opin.sym`, name it `out`
- VSS: `iopin.sym`, name it `VSS`
- VDD: `iopin.sym`, name it `VDD`

These pins will attach to the nets of the same name, so we dont need to connect them further.

Your schematic should something look like this:



Creating the Symbol

Now, we want to create a symbol for our inverter.

For this, navigate to the `Symbol` menu in the toolbar and click on `Make symbol from schematic`. This will create a `inverter.sym` file in the same directory as `inverter.sch`.

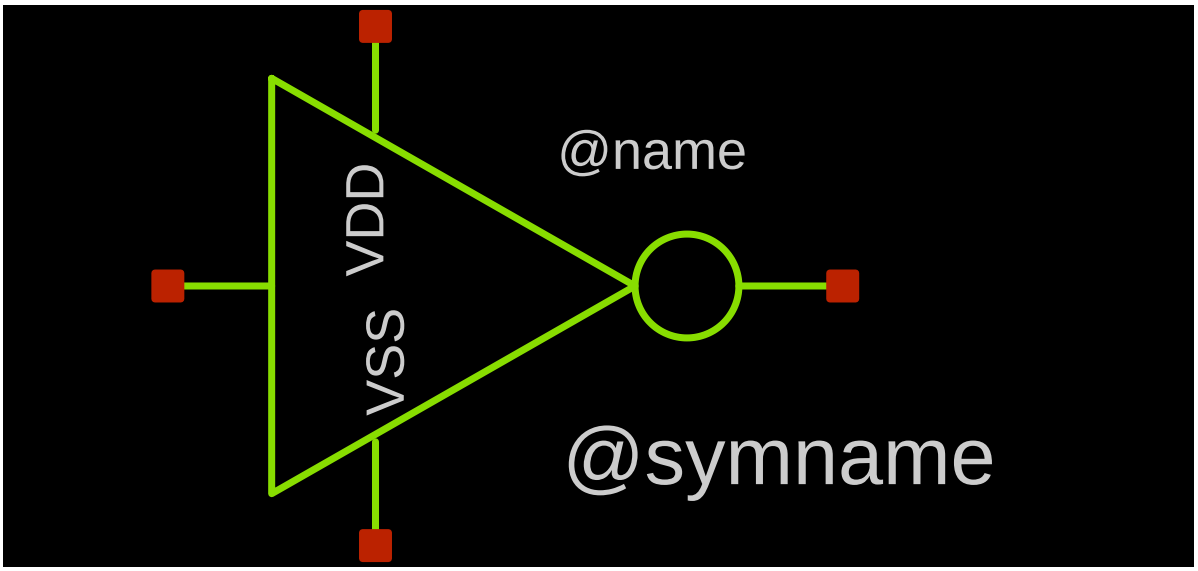
Open the symbol using

```
xschem inverter.sym
```

You should see the default generated symbol for the inverter.

This is good but it looks quite boring. You can improve it by editing the symbol by moving/rotating the pins around and by adding new polygons (`P`), generally using the tools in the toolbar.

An improved version could look like this:



Creating the Testbench

Now, we create a testbench for our inverter.

For this, create the testbench schematic:

```
xschem inverter_tb.sch
```

Placing our new Block

Place your inverter symbol by going to the current directory via the `Current Dir` button of the `Choose symbol` dialog (`Insert`) and selecting the `inverter.sym` file.

Placing the Testbench Components

Now, we need to connect the input and output of the inverter to the testbench. Specifically, we add:

- Input: Rectangle voltage source
- VSS/VDD: DC supply voltages
- Output: Some loading resistor

Place three voltage sources `vsource` from the generic library.

Place one resistor `res` from the generic library.

Place grounds (`gnd`) at the lower terminals of each voltage source and the resistor.

Connect the voltages and resistor accordingly using wires (`W`). I suggest using labels for clarity.

Set the voltages for the voltage sources:

- VSS: `0`
- VDD: `1.5`
- Input: `"pulse(0 1.5 1n 0.1n 0.1n 1n 2n)"` (**Dont** forget the quotes)
  This generates a pulse waveform from 0 to 1.5V, with a delay for the first edge of 1 ns, 0.1ns fall and rise times, 1 ns high time and 2 ns period.

Set the resistance to `1Meg` .

Setting up the Simulation Commands

Now, we need to write the commands for actually running the simulation. We will just take them from the XSCHEM PDK starting site. In a new terminal, open XSCHEM via `xschem` . From the PDK start page, select (select multiple via holding `Shift`):

- "Load IHP SG13G2 spice models for ngspice" (`Libs_Ngspice`)

- "Simulation skeleton for ngspice" (`SimulatorNGSPICE`)
- "Create .save file, create netlist, simulate with ngspice" (`SimulateNGSPICE`)

Copy these blocks via `Ctrl+C` and paste them into your testbench schematic via `Ctrl+V` .

At the moment, this simulates the operating point only. We need to modify the `SimulatorNGSPICE` code block, specifically, we need to change the `op` command to (to also save all voltages/currents):

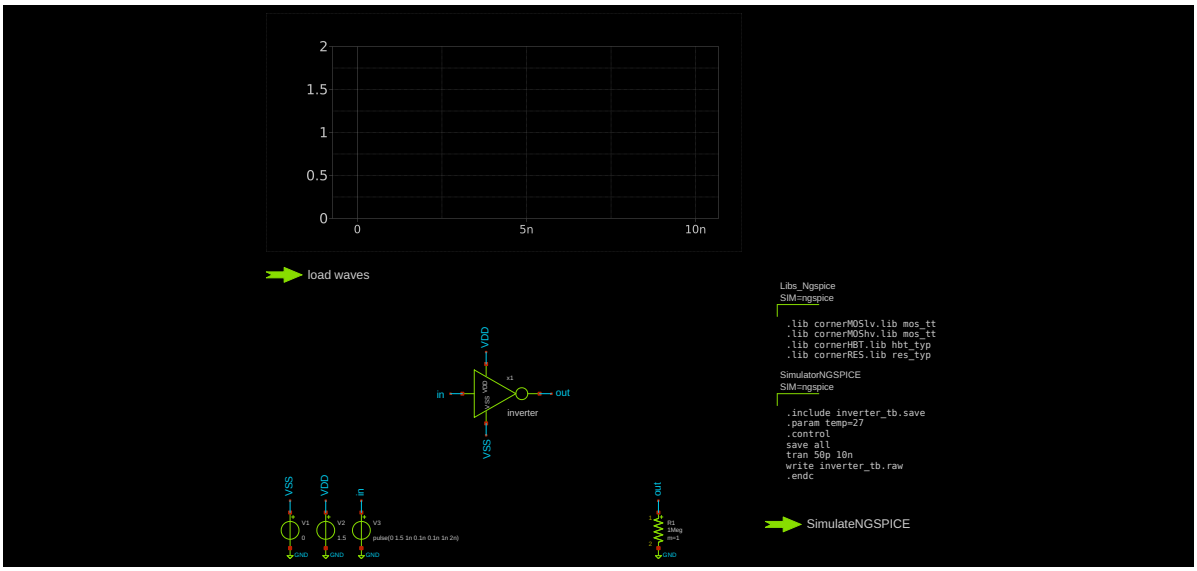
```
save all
tran 50p 10n
```

While you are in the code block, also change all `<filename>` instances to `inverter_tb` .

Adding Graphs

We also want a graph, which can be placed via `Simulation -> Graphs -> Add waveform graph` . Also, add a waveform reload launcher via `Simulation -> Graphs -> Add waveform reload launcher` .

Your testbench should look something like this:



Running the testbench

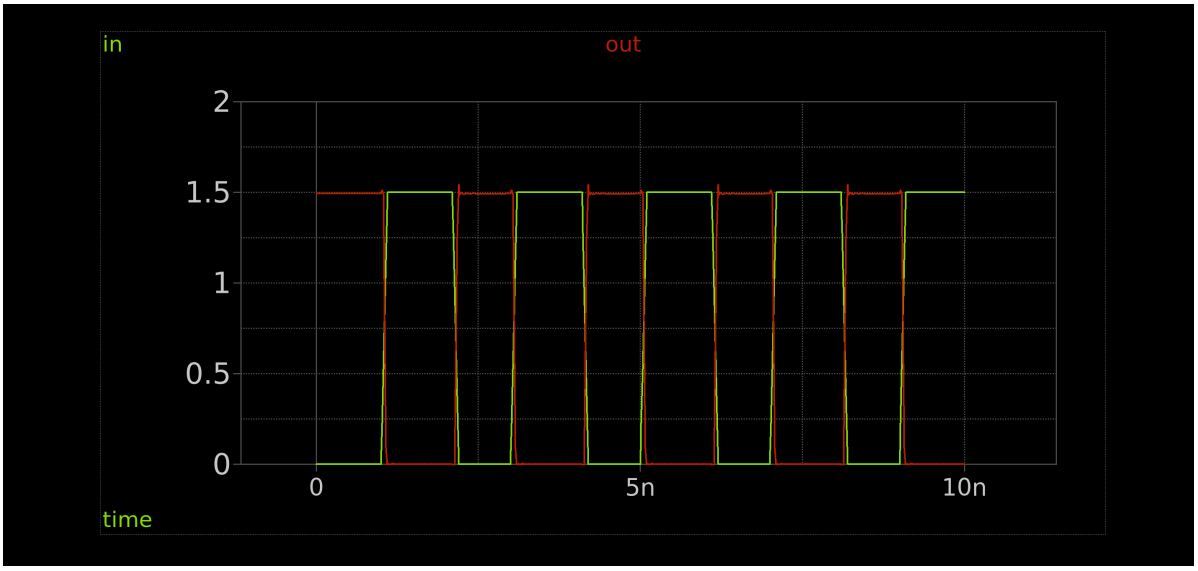
Now, run the testbench via `Ctrl + left click` on the `SimulateNGSPICE` launcher. Then, load the waves via the `load waves` launcher.

Now, we want to add some traces to the graph. You can do this by double-clicking in the middle of the graph. You should be able to see a list of traces to plot on the left side. There, find and add the `in` and `out` traces by double-clicking on them.

By default, all traces are the same color. To changes that, click on the `AUTO SET` button on the bottom right.

Now, click `Apply` and `OK`. You should see some traces in the graph, but you need to adjust the time scale to see the pulses clearly. You can do this by holding `Shift` and scrolling the mouse wheel to zoom in and out, or by holding `Ctrl` and scrolling to pan.

Your plot may look something like this:



Analysis in Python

Sometimes, doing analysis of the simulation results can be cumbersome. To facilitate this, we can use Python scripts to process the raw data files generated by the simulations. This allows us to automate the analysis and generate plots or reports more easily.

Reading `.raw` files

For reading in `.raw` files, multiple libraries exist. For me, it was easiest to use my own file parser:

```
class NGSpiceRaw:
    def __init__(self, fname: str, live: bool = True):
        """
        Initialize the NGSpiceRaw object.

        Args:
            fname (str): The name of the .raw file to read.
            live (bool): Whether to reload the file on each access.
        """
        self.fname = fname
        self.live = live

        self._reload()

    def _reload(self):
        self.arrs, self.plots = self._read_raw(self.fname)
        self.plot, self.arr = self.plots[-1], self.arrs[-1]
        print(f"Loaded {len(self.plots)} plots from {self.fname}")

    def _read_raw(self, fname: str):
        """
        Read a binary ngspice .raw file.

        Returns:
          arrs  : list of numpy structured arrays, one per plot
          plots : list of metadata dicts, parallel to arrs
        """
        with open(fname, 'rb') as fp:
            arrs = []
            plots = []
            plot = {}
            while True:
                line = fp.readline(BSIZE_SP)
                if not line:
                    break
                parts = line.split(b':', 1)
                if len(parts) != 2:
                    continue
                key, val = parts[0].lower(), parts[1].strip()
                if key in MDATA_LIST:
                    plot[key] = val
                if key == b'variables':
                    nvars   = int(plot[b'no. variables'])
                    npoints = int(plot[b'no. points'])
                    plot['varnames'] = []
                    plot['varunits'] = []
                    for _ in range(nvars):
                        ascii_line = fp.readline(BSIZE_SP).decode('ascii')
                        idx, name, *unit = ascii_line.split()
                        plot['varnames'].append(name)
                        plot['varunits'].append(unit[0])
                if key == b'binary':
                    # build dtype (complex if flagged, else float)
                    fmt = np.complex_ if b'complex' in plot[b'flags'] else float
```

```
                    row_dtype = np.dtype({
                        'names':   plot['varnames'],
                        'formats': [fmt]*len(plot['varnames'])
                    })
                    # read data block
                    data = np.fromfile(fp, dtype=row_dtype, count=npoints)
                    arrs.append(data)
                    plots.append(plot.copy())
                    plot.clear()
                    fp.readline()

        return arrs, plots

    def select(self, idx: int):
        """
        Select a plot by index.
        """
        if idx < -len(self.plots) or idx >= len(self.plots):
            raise IndexError("Index out of range")

        self.plot = self.plots[idx]
        self.arr = self.arrs[idx]

        return self.plot, self.arr

    @property
    def names(self):
        return self.arr.dtype.names

    def __getitem__(self, key):
        """
        Get a variable by name or index.
        """
        if self.live:
            self._reload()

        if key in self.names:
            return self.arr[key]
        else:
            raise KeyError(f"Variable '{key}' not found")

    def __setitem__(self, key, value):
        """
        Set a variable by name or index.
        """
        if self.live:
            self._reload()

        if key in self.names:
            raise KeyError(f"Variable '{key}' already exists")
        else:
            # Add new variable to the array
            new_dtype = np.dtype(self.arr.dtype.descr + [(key, value.dtype)])
            new_arr = np.zeros(self.arr.shape, dtype=new_dtype)
```

```
            for name in self.names:
                new_arr[name] = self.arr[name]
            new_arr[key] = value
            self.arr = new_arr
            self.arrs[-1] = new_arr
            self.plot['varnames'].append(key)
            self.plot['varunits'].append('')
            self.plot[b'no. variables'] = str(len(self.plot['varnames']))
            self.plot[b'no. points'] = str(len(self.arr))
```

This parser is quite easy to use:

```
data = NGSpiceRaw("../simulations/dc_lv_nmos.raw", live=False)
print("Fields:", data.names)

v_ds = data["v(v-sweep)"]
i_vd = data["i(vd)"]
v_th = data["v(@n.xm1.nsg13_lv_nmos[vth])"]
gm = data["@n.xm1.nsg13_lv_nmos[gm]"]
gds = data["@n.xm1.nsg13_lv_nmos[gds]"]
cgs = data["@n.xm1.nsg13_lv_nmos[cgsol]"]
cgd = data["@n.xm1.nsg13_lv_nmos[cgdol]"]
```

Sweeps

I did not yet find out how to properly extract sweeps from the data. Here is the hacky workaround I have used:

```
class Sweep:
    def __init__(self):
        self.values = np.array([])

    def linear(self, start: float, stop: float, step: float):
        self.values = np.arange(start, stop + step, step)

        return self

    def split(self, data: np.ndarray):
        if len(data) % len(self.values) != 0:
            raise ValueError("Data length is not divisible by n")
        return np.array(np.array_split(data, len(self.values)))

    def __getitem__(self, index: int):
        if index < 0 or index >= len(self.values):
            raise IndexError("Index out of range")
        return self.values[index]
```

Which can be used like

```
vgs_sweep = Sweep().linear(0.0, 1.0, 0.1)

v_ds = vgs_sweep.split(v_ds)[0] # This will be the same every time since it's the
second sweep parameter
i_vd = vgs_sweep.split(i_vd)
v_th = vgs_sweep.split(v_th)
gm = vgs_sweep.split(gm)
gds = vgs_sweep.split(gds)
cgs = vgs_sweep.split(cgs)
cgd = vgs_sweep.split(cgd)
```

Plotting

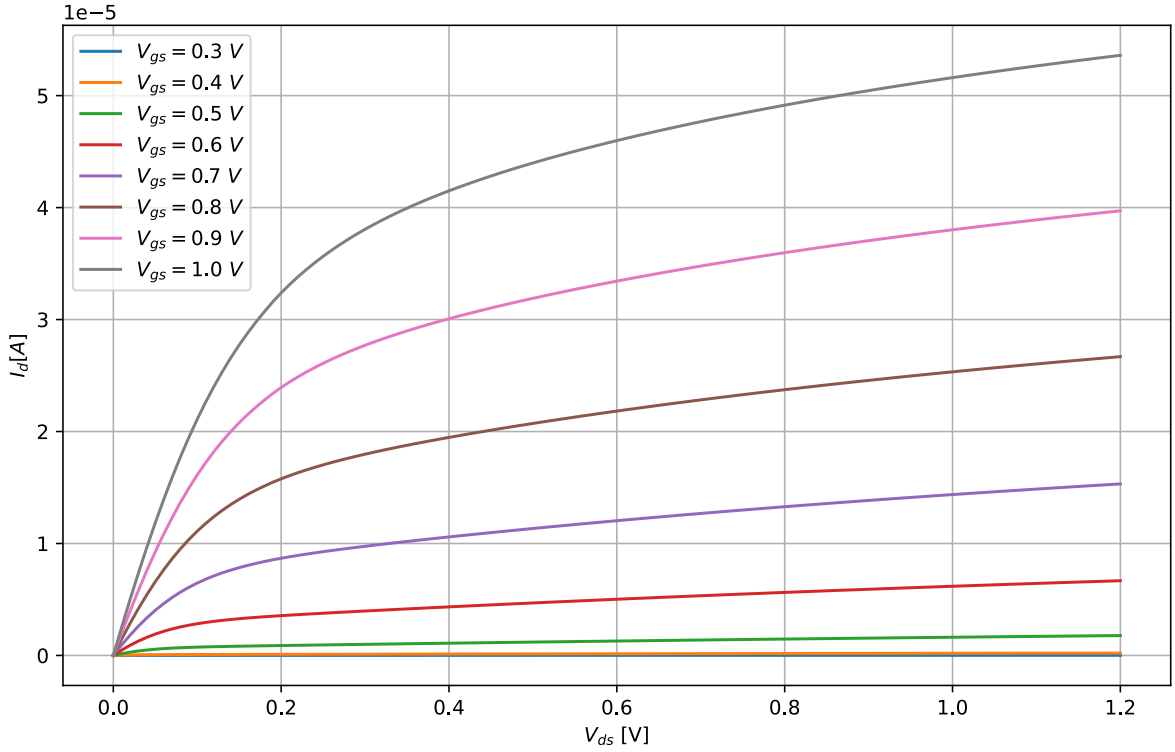
To plot the resulting drain current of this sweep of a MOSFET, you would then do:

```
plt.figure(figsize=(10, 6))

for i, (vgs, values_split) in enumerate(zip(vgs_sweep.values, i_vd)):
    plt.plot(v_ds, values_split, label=f"$V_{{gs}} = {vgs:.1f}\\ V$",
linewidth=1.5)

plt.xlabel("$V_{ds}$ [V]")
plt.ylabel("$I_{d} [A]$")
plt.legend()
plt.grid(True)
plt.show()
```

Which results in the following plot:



Introduction

How does co-simulation work in ngspice?

Cosimulation with ngspice enables mixed-signal simulation by combining SPICE-based analog circuit models with a digital HDL simulation engine. In this setup, ngspice runs the analog portion of the design (e.g., transistor-level circuits, passive networks), while the digital portion (e.g., control logic, ADC state machines) is executed by an external HDL simulator. Communication occurs via a well-defined interface - ngspice exchanges signal values with the digital simulator at synchronized timesteps, ensuring that analog and digital events remain time-aligned.

When using Icarus Verilog (iverilog and vvp), ngspice interacts through its d_cosim or d_process model, spawning the Icarus Verilog simulation process and passing data through pipes or sockets. Compared to Verilator, which compiles Verilog into a C++ model that must be linked and rebuilt for every change, Icarus Verilog offers faster iteration, direct interpretation of HDL without a compile-link cycle, and better compatibility with ngspice's cosimulation interface - making it the preferred choice when doing rapid mixed-signal development and debugging.

Creating a Mixed-Signal Simulation

To get a full mixed-signal simulation, we need:

- Digital design, we write it in SystemVerilog
- Analog design, in SPICE, designed in XSCHEM
- A symbol representing the digital design as a block for XSCHEM
- Testbench, combining both Analog and Digital designs and defining their interaction

In our example, we create a simple PWM DAC, with digitally adjustable output voltage levels.

Digital Design

We create a simple SystemVerilog module with the following signals:

Inputs

- `clk_i` : Clock signal
- `rst_ni` : Active-low reset signal
- `set_i[3:0]` : 4-bit control signal for the duty cycle

Outputs

- `pwm_o` : PWM output signal

The module might look something like this:

```
`timescale 1ns/1ps

module pwm_dac (
    input logic       clk_i,  // Clock signal
    input logic       rst_ni, // Active-low reset signal
    input logic [3:0] set_i,  // 4-bit control signal for the duty cycle

    output logic      pwm_o   // PWM output signal
);

    logic [3:0] counter_d, counter_q;

    initial begin
        $display("PWM DAC digital part started");
        $dumpfile("pwm_dac.vcd");
        $dumpvars(0, pwm_dac);
    end

    always_ff @(posedge clk_i or negedge rst_ni) begin
        if (!rst_ni) begin
            counter_q <= '0;
        end else begin
            counter_q <= counter_d;
        end
    end

    always_comb begin
        if (!rst_ni) begin
            counter_d = '0;
        end else begin
            counter_d = counter_q + 1;
        end
    end

    assign pwm_o = (counter_q < set_i) ? 1'b1 : 1'b0;

endmodule
```

Important: The `` `timescale `` directive is crucial for enabling cosimulation.

We save this code as `pwm_dac.sv` and compile it with Icarus Verilog. This is done with the following command:

```
iverilog -g2012 -o pwm_dac pwm_dac.sv
```

The `-g2012` flag tells Icarus Verilog to use the SystemVerilog 2012 standard.

This gives us a compiled simulation model that we can use for testing. You can actually open the resulting `pwm_dac` file with your text editor and see that it's a script for the `vvp` tool.

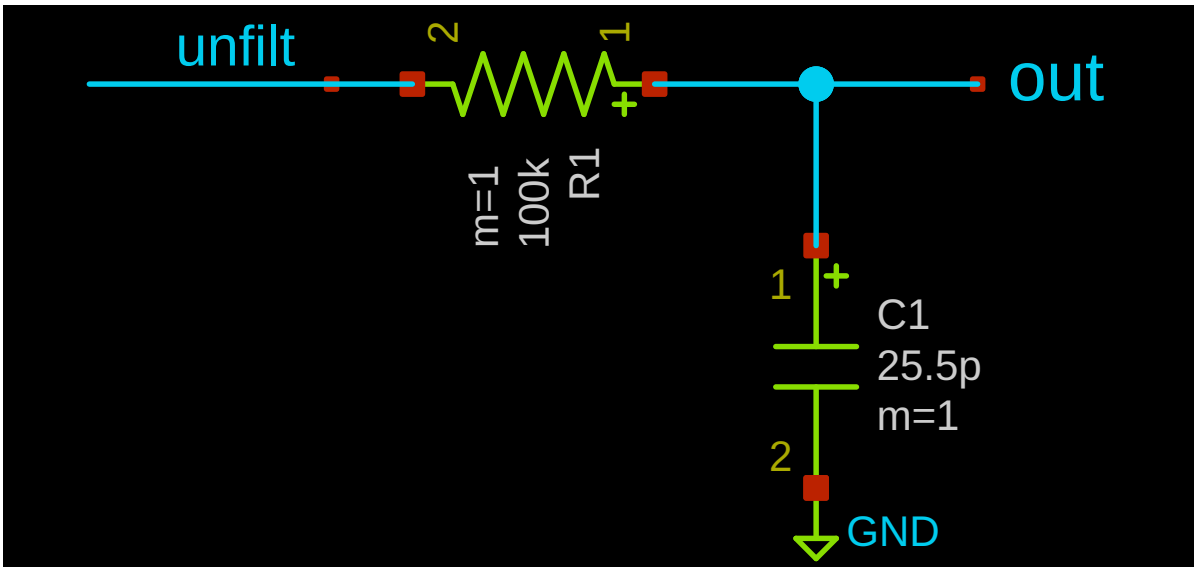
Analog Design

We design a simple 1-stage RC lowpass filter. We assume the input frequency to be $100\ MHz$. The resulting output frequency is thus $\frac{100\ MHz}{16} = 6.25\ MHz$. We put the corner frequency of the filter two decades lower, at $62.5\ kHz$. With $R = 100\ k\Omega$, we need $C = \frac{1}{2\pi f_c R} \approx 25.5\ pF$.

We implement the circuit in the testbench `pwm_dac_tb.sch` directly via

```
xschem pwm_dac_tb.sch
```

The resulting circuit looks something like this:



Symbol for the Digital Design

We need to create a symbol for the digital design that we can use in the testbench. This symbol should represent the inputs and outputs of the `pwm_dac` module. Create a new symbol with

```
xschem pwm_dac.sym
```

Directly after opening, please save the symbol via `File -> Save as symbol` (or `Ctrl + Alt + S`). This ensures that the symbol properties are properly setup.

Now, we first configure these properties by double-clicking the emtpy canvas. This opens an empty text input where we enter:

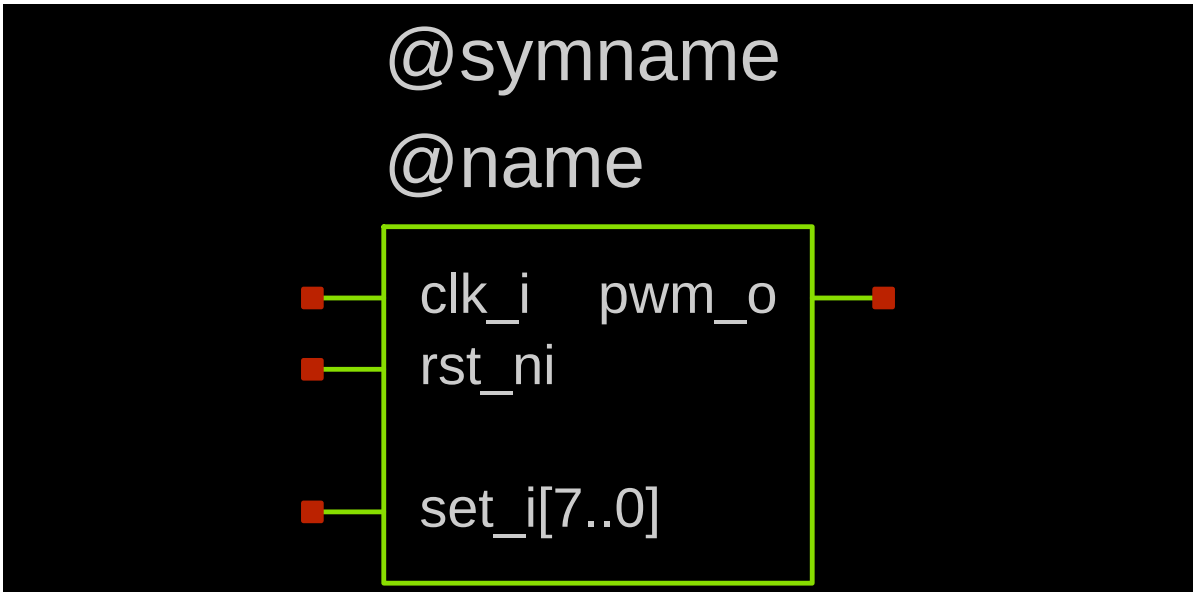
```
type=primitive
format="@name [ @@clk_i @@rst_ni @@set_i[3..0] ] [ @@pwm_o ] @model"
template="name=A1 model=pwm_dac"
```

This defines:

- The type of the symbol (`primitive`, other options include `subcircuit` for actual circuits)
- The format of the circuit. This has to be very specific:
  a. `@name`: The name of the instance in the schematic (e.g., `A1`)
  b. `[ @@clk_i @@rst_ni @@set_i[3..0] ]`: The input ports of the instance
  c. `[ @@pwm_o ]`: The output ports of the instance
  d. `@model`: The model name of the instance (e.g., `pwm_dac`)
- The template for the instance. This defines how the instance will be instantiated in the schematic. Here, the instance gets the name `A1` and the model `pwm_dac`, which we have built above.

Now, we place the pins of the symbol in the schematic. Do this via `Symbol -> Place symbol pin` (`Alt + P`). Make sure to set the correct name and direction (`in`, `out` or `inout`) for each pin.

Your symbol might look something like this:



The `@symname` and `@name` texts are placeholders that will be replaced with the actual symbol and instance names when the schematic is instantiated. You can leave them out.

Testbench

Now, we go back to the testbench

```
xschem pwm_dac_tb.sch
```

Place our `pwm_dac.sym` symbol into the schematic. Now, we need to specify the cosimulation model for the PWM DAC. We can do this in this symbols properties directly. When you open the symbols properties by double-clicking it, you should be able to see the name and model properties. On a new line, add the following:

```
device_model=".model pwm_dac d_cosim simulation=\"ivlng\" sim_args=
[\"../pwm_dac\"]"
```

The `device_model` property specifies the model name and the simulation parameters for the PWM DAC. This tells ngspice to use the `ivlng` simulation method with the specified arguments.

Now we can hook up the output up to the lowpass filter. Also place voltage sources to input the clock and reset signals, as well as the control inputs. Single signals from a bus can be accessed via the `a0 up to a3` labels if you have a `a[3..0]` label.

The control inputs `set_i0 to set_i3` are each connected to their own voltage source so that we can simulate a simple changing digital input. A simple example would be the following:

- `set_i0` : Constant 1.5V
- `set_i1` : Constant 1.5V
- `set_i2` : Step 0V to 1.5V at 15us
- `set_i3` : Constant 0V

With this, we simulate a simple step from input code 0x03 to 0x07 at 15us, such that we can observe the behavior of the PWM DAC.

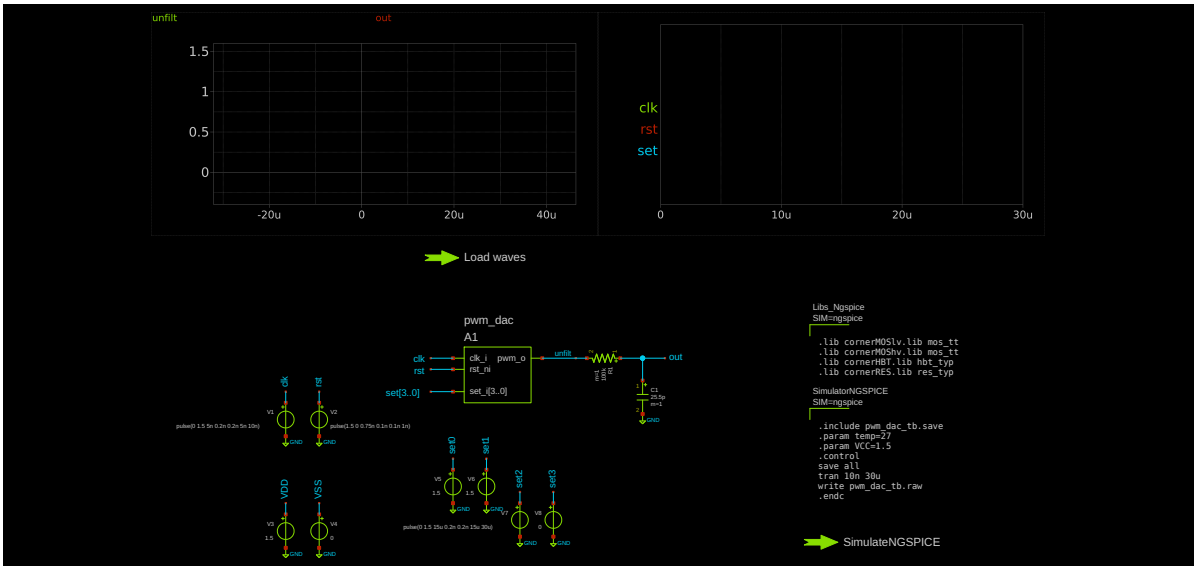
Add the device models and simulation commands/launcher. For the simulation commands, we use the following:

```
.include pwm_dac_tb.save
.param temp=27
.param VCC=1.5
.control
save all
tran 10n 10u
write pwm_dac_tb.raw
```

Here, the important part is `VCC`. This defines the logic levels for the digital->analog and analog->digital bridges. These bridges can be set up more sophisticated (See Bridges), but for now we just set `VCC` to our vdd, 1.5V. Also, we set the simulation time to 10 microseconds with a timestep of 10 nanoseconds and to save all data to `pwm_dac_tb.raw`.

Also add a waveform loader and two graphs.

The testbench might look like this:



Simulation

Currently, for the simulation to run successfully, some tweaks need to be made when you are using the IIC-OSIC-TOOLS container:

```
sudo mkdir -p /usr/local/lib/ngspice
sudo cp /foss/tools/ngspice/lib/ngspice/ivlng.vpi /usr/local/lib/ngspice
sudo cp /foss/tools/iverilog/lib/libvvp.so /foss/tools/ngspice/lib/ngspice
```

We can then run the simulation with the simulation launcher. You should see such an output:

```
******
** ngspice-44.2 : Circuit level simulation program
** Compiled with KLU Direct Linear Solver
** The U. C. Berkeley CAD Group
** Copyright 1985-1994, Regents of the University of California.
** Copyright 2001-2024, The ngspice team.
** Please get your ngspice manual from https://ngspice.sourceforge.io/docs.html
** Please file your bug-reports at http://ngspice.sourceforge.net/bugrep.html
** Creation Date: Tue Jul 29 08:21:22 UTC 2025
******

Note: No compatibility mode selected!

Circuit: ** sch_path: /workspaces/oscic-playground/cosim/pwm_dac_tb.sch

Note: No compatibility mode selected!

Reducing trtol to 1 for xspice 'A' devices
Doing analysis at TEMP = 27.000000 and TNOM = 27.000000

Using SPARSE 1.3 as Direct Linear Solver

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

Node                    Voltage
----                    -------
out                     0
unfilt                  0
clk                     0
rst                     1.5
vdd                     1.5
vss                     0
set0                    1.5
set1                    1.5
set2                    1.5
set3                    0
v#branch                0
v7#branch               0
v6#branch               0
v5#branch               0
v4#branch               0
v3#branch               0
v2#branch               0
v1#branch               0
auto_dac#branch_1_0     0

PWM DAC digital part started
VCD info: dumpfile pwm_dac.vcd opened for output.
 Reference value 9_95157e-06
No. of Data Rows : 205255
binary raw file "pwm_dac_tb.raw"
ngspice 1 ->
```

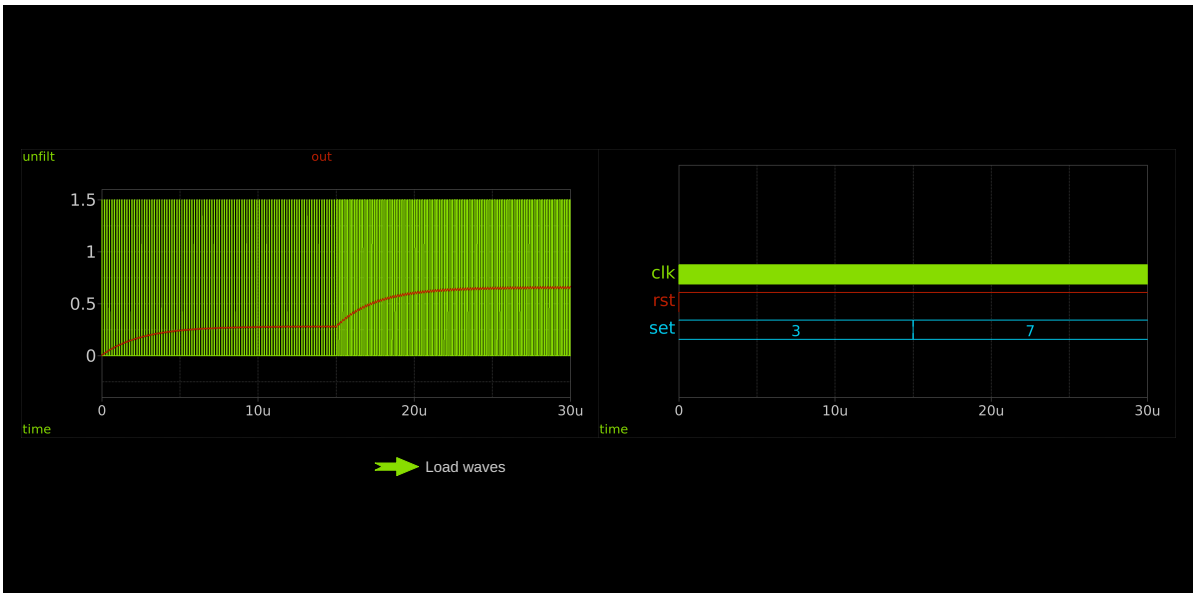
You can see the `PWM DAC digital part started` and `VCD info: dumpfile pwm_dac.vcd opened for output.` which indicate that the digital part of the simulation is running properly!

Visualization

XSCHEM's built in graphs already include a handy way to display digital and analog signals concurrently. We use one of our graphs for the digital signals and one for the analog signals.

In the graph for the analog signals, add the `unfilt` and `out` signals. The settings can remain the default.

In the graph for the digital signals, add the `clk`, `rst` signals as well as `set;set3,set2,set1,set0`, which allows us to look at the `set` signals as a bus. Your plots might look like this:



We can see that our simple PWM DAC example works as expected.

Now, you can also open the .vcd file in a waveform viewer to inspect the digital part more closely.

Bridges

What are bridges?

Bridges in ngspice define the transitions between the analog and digital domain. There are three different types of bridges:

- `dac_bridge`: Digital-to-Analog node bridge, at outputs of digital blocks
- `adc_bridge`: Analog-to-Digital node bridge, at inputs of digital blocks
- `bidi_bridge`: Bidirectional bridge, allowing for two-way communication between analog and digital domains

These bridges have a variety of configurable parameters, such as trigger voltage levels, rise/fall times and delays.

Automatically placed Bridges

By default, ngspice inserts some bridges automatically between analog and digital domains based on the circuit topology. The default bridges are:

- Digital-to-Analog: `.model auto_dac dac_bridge(out_low = 0 out_high = 'VCC')`
- Analog-to-Digital: `.model auto_adc adc_bridge(in_low = 'VCC/2' in_high = 'VCC/2')`
- Bidirectional: `.model auto_bidi bidi_bridge(out_high='VCC' in_low='VCC/2' in_high='VCC/2')`

But they can be overwritten. I suggest using bidirectional bridges for all of them, since they have the most flexible parameters. You can change the default bridges as follows:

```
.model adc_buff_clk adc_bridge(in_low = 'vdd/2' in_high = 'vdd/2')

.control
.model adc_buff_clk adc_bridge(in_low='vdd/2' in_high='vdd/2')

.control
pre_set auto_bridge_d_out =
+ ( \".model auto_bridge_out bidi_bridge(direction=0 out_high='vdd' t_rise=0.2n
t_fall=0.2n)\"
+   \"auto_bridge_out%d [ %s ] [ %s ] null auto_bridge_out\" )
pre_set auto_bridge_d_in =
+ ( \".model auto_bridge_in bidi_bridge(direction=1 in_low='vdd/3'
in_high='vdd/3*2')\"
+   \"auto_bridge_in%d [ %s ] [ %s ] null auto_bridge_in\" )
.endc
```

This defines the following models:

- `auto_bridge_out`: A bidirectional bridge for analog-to-digital conversion.
- `auto_bridge_in`: A bidirectional bridge for digital-to-analog conversion.
- `adc_buff_clk`: A bridge for clock signals in ADCs. If you have a hysteresis on the digital inputs, your signal may be undefined for some duration of the clock edge. Since one may not want this behavior in their clock inputs, we define a bridge model specifically for clock inputs.

Manually placing Bridges

Bridges can also be placed manually. The components for the bridges are located in the generic library and are called `adc_bridge.sym` and `dac_bridge.sym`.

We can take our clock buffer above as an example. Place the `adc_bridge.sym` directly in front of the clock pin of a digital design. Double-click on the bridge and change the model from `adc_buff` to `adc_buff_clk`. Now, the clock input should use the properties as described in the

`adc_buff_clk` model.