Introduction

Welcome to the exercise lessons for the electronics for physicists II course. Today's exercise will consist of first installing the necessary software to simulate our digital designs and then trying out with a few examples of combinatorial logic.
The software runs on Windows 10 and 11 as well as Linux (Ubuntu 24.04). The simulations tools also run on Mac (Apple Silicon). The FPGA development environment works with some restrictions on Mac.

Task 0: Setting up the software

Windows users

Setup Windows Subsystem for Linux (WSL2) and the software for Linux:

1. Open the power shell as administrator
2. Type `wsl --install -d Ubuntu-24.04`
3. Reboot Windows
4. After the installation finished, you can start the WSL app on windows. You should see the typical Linux console. Make a symbolic link to your Windows home directory as `ln -s /mnt/c/Users/<your Windows user name> home_win`. This way, you can easily access the same files from Windows and Linux.
5. Download `install_simulator_Linux_WSL.sh` from the `Software/Simulator/Linux_WSL` folder from moodle.
6. Open a console and navigate into the directory where you downloaded the installer script to. Make the script executable: `chmod +x install_simulator_Linux_WSL.sh`
7. Install all the needed packages with `./install_simulator_Linux_WSL.sh`. Don't run this script as sudo, this is handled internally.
8. Install vscode (from `https://code.visualstudio.com/download`) In vscode: install plugin for SystemVerilog (SystemVerilog - Language Support)

Linux users

Setup the open source software on Linux (it only works on Ubuntu 24.04):

1. Download `install_simulator_Linux_WSL.sh` from the `Software/Simulator/Linux_WSL` folder from moodle.
2. Open a console and navigate into the directory where you downloaded the installer script to. Make the script executable: `chmod +x install_simulator_Linux_WSL.sh`
3. Install all the needed packages with `./install_simulator_Linux_WSL.sh`. Don't run this script as sudo, this is handled internally.
4. Install vscode (from `https://code.visualstudio.com/download`) In vscode: install plugin for SystemVerilog (SystemVerilog - Language Support)

Mac users

Setup the open source software on Mac:

1. If not yet installed: install brew (see `https://brew.sh/`). You find the installer on moodle (`Software/Simulator/Mac`). Please run this installer.
2. If not done yet: Add the following to `~/.zshrc`: `export PATH=$PATH:/opt/homebrew/bin`. After that, it is recommended to close and re-open the terminal.
3. Download `install_simulator_mac.sh` from the `Software/Simulator/Mac` folder from moodle.
4. Make the script executable: `chmod +x install_simulator_mac.sh`
5. Perform the installation: `./install_simulator_mac.sh`. Don't run this script as sudo, this is handled internally.
6. Install vscode (from `https://code.visualstudio.com/download`) In vscode: install plugin for SystemVerilog (SystemVerilog - Language Support)

Task 1: Is everything working?

To test if everything went well you can try out a few programs. This will also help you to get familiar with using the Linux UI.

Virtual environment for our Python based tools

On Linux (or WSL) we created a *virtual environment* for the Python software. Before you can use it, you need to *activate* it with:

```
source ~/efp/bin/activate
```

If the environment is active you find `(efp)` in front of your user name on the console.

To deactivate, just type `deactivate`.

First simulation

A big part of the work of Hardware development is doing simulations. In Electronics for Physicists 2 we will use cocotb, a python package that allows you to simulate SystemVerilog. We have prepared a small module you can simulate and analyze the results of. For the simulation to work you need to activate our virtual environment.

1. Download this week's exercise files and unzip it. Go into the directory for task 1. You should have a makefile, a System Verilog (.sv) file and a Python file.
2. Take a look at the Python file, can you figure out what inputs we will give to our module? Ask the assistants! (*Hint: our module has two inputs a_in, b_in and one output c_out*)

3. Open the terminal and navigate to the folder containing all the files (in the terminal you can use `cd <foldername>` to enter a folder and `cd ..` to go up one folder)
4. Run `make`. The simulation should now start and be done withing a few seconds
5. Run `gtkwave dump.vcd`, this should open the waveform viewer that we will use to look at our different signals.
6. On the top left of the window you should see an item called *gate*. After clicking it some signals called a_in, b_in, c_out and clk_i should appear below.
7. Double click the signals a_i, b_i and c_o, to view their waveforms. Can you figure out what logic gate we implemented? Take a look at `test.sv`, to confirm.

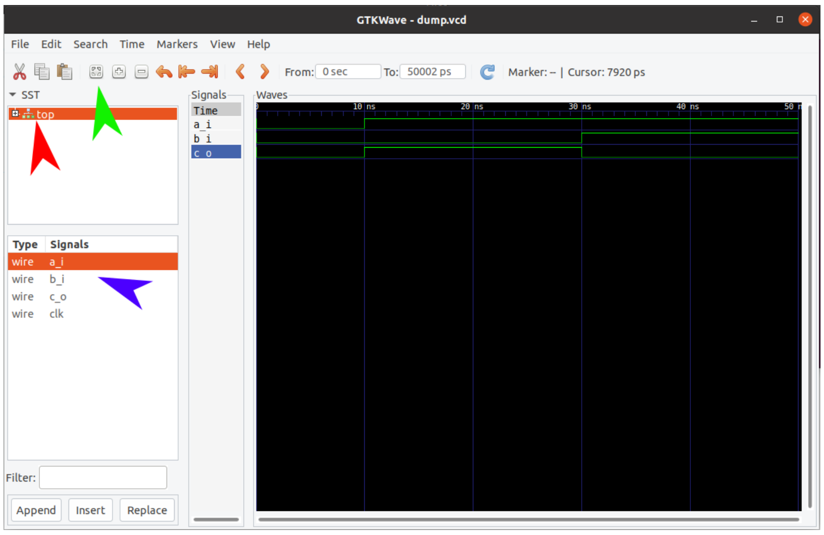


Figure 1: Screenshot of GTKWave. First click *gate* next to the red arrow. Then double click on the signal names next to the blue arrow to add the display. The green arrow points at the auto-zoom button, which is very useful.

Short introduction to SystemVerilog

We are now ready to dive into the world of hardware description languages (HDLs). As you have heard in the lecture today there are two main types of circuits/logic we will be dealing with, combinational and sequential. In today's exercise we will be dealing with combinational circuits. Combinational logic can be formed by logic gates and look up tables, without the use of registers. As soon as one of the inputs changes the output will change (neglecting propagation delay).
The language we will be using to describe hardware is SystemVerilog. In SystemVerilog there are two main ways to describe combinational circuits: `assign` statements and `always_comb` blocks.

```
// assign statement
assign a = b && c;

// always_comb block
```

```
  always_comb begin
      a = b && c;
  end
```

In both cases we describe a simple and gate. While you could achieve the same results with both syntaxes, for style reasons you should use `assign` only for connecting two signals or very simple logic. For more complicated logic use `always_comb`. Also notice that in the code the block is defined with `begin` and `end` these take the job of { from C or indentations from python when defining blocks.
Another very important concept you will need in today's exercise are modules. Modules you can imagine as circuits you predefine and can instantiate and connect to your signals. In a way they are similar to functions in programming languages, but don't forget they are **not** the same! We are describing hardware and not a sequential program. So a better way to think about them are predifined building blocks that you can connect up to your design. They usually have input and output ports that connect the inside logic of the module to the outside. To define a module you can use the following syntax:

```
  module mymodule(
        // here we define the connections going in and out of our module
        input logic a_i, // 1 bit input a_i
        input logic b_i, // another 1 bit input named b_i
        output logic c_o, // a 1 bit output named mysignalname_o
        output logic[3:0] myarray_o // 4 bit wide signal
        );

        // describe the logic of your module here

        logic internal; // of course you can also define internal signals

        assign internal = a_i;
        assign c_o = internal | b_i; // internal OR b_i
        assign myarray_o = {a_i, a_i, b_i, b_i}; // concatenate multiple signals

endmodule
```

Now that we have defined our module we can instantiate it in another module (possibly in a different file). In this case the module top contains an instance of mymodule, which we call peter:

```
  module top(
        input logic one_i,
        input logic two_i,
        output logic three_o,
        );

        logic[3:0] four;

        mymodule peter(
              .a_i(one_i),
              .b_i(two_i),
              .c_o(three_o),
              .myarray_o(four)
        );

endmodule
```

If you want to write nice code, which is strongly recommended if you have a bigger project, you could take a look at the Institute for Integrated Systems's style guide, here. It is also the reason why we use `_i` in the names of inputs to a module and `_o` in the names of outputs of a module. This is only a naming convention. (The `input` and `output` statements on the other hand are mandatory.)

If something is unclear or you just want to know more about a certain topic please ask one of the teaching assistants.

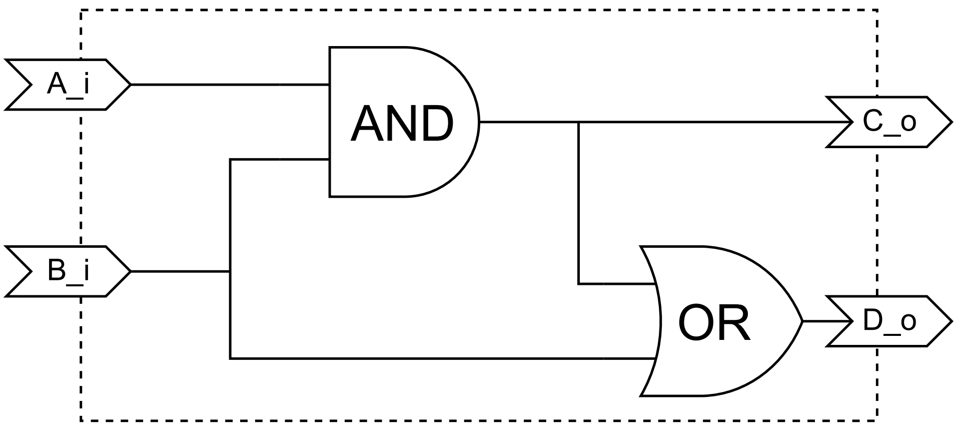
Task 2: Your first module

Now that you have read about the basics of combinational logic in SystemVerilog try writing your own very simple module.

a) Take a look at the following circuit and write down the truth table for the module.

b) If you have not done so yet, download the files for this task from moodle. In the file `circuit.sv` you can find a template written in SystemVerilog. Complete the template to implement the circuit from above. *Note: There are many different ways to do this.*

c) Now let's see if your design works. For this use the testbench in file `testbench_circuit.py`. It applies signals to the inputs of the device-under-test (DUT) and records the outputs. Take a look at the file and try to understand what it does. You can start the testbench by using the `make` command. (*Note: You have to be in the correct folder the Makefile is in for the testbench to run.*). Use `gtkwave dump.vcd` to look at the waveforms the testbench produced. Is your circuit working as expected?



0 0 | 0 0
0 1 | 0 1
1 0 | 0 0
1 1 | 1 1

Figure 2: The circuit diagram for task 1. A module is depicted with inputs `A_i`, `B_i` and outputs `C_o`, `D_io`.