ECE210 TinyTapeout Sky130 Template – Setup & Customization Guide

This guide walks you through:

- **Setting up the Docker-based development environment**
- **Running the provided example simulation to verify your setup**
- **Creating and testing your own design**
- **Understanding the project directory structure**

It assumes basic familiarity with Git, GitHub, and a Unix-like shell.

---

1. Prerequisites

Before you begin, make sure you have:

- **Git** installed and configured (`git --version`)
- **Docker** installed and running (`docker --version`). If you do not have it, follow the official installation guide: **Get Docker**
- A **GitHub account**

No local Verilog or cocotb installation is required; everything runs inside Docker.

---

2. Create Your Own Repository From the Template

1. **Open the template repo in your browser**
   Go to the TinyTapeout template:
   - `https://github.com/TinyTapeout/ttsky-verilog-template`
2. **Create your own repository using this template**
   - Click **"Use this template"** on GitHub.
   - Set a meaningful name, e.g. **`ECE210-ttsky`**.
3. **Clone your new repository locally**

```
cd Path/to/your/directory
git clone git@github.com:<your_username>/ECE210-ttsky.git
cd ECE210-ttsky
```

Replace `<your_username>` and the repository name as appropriate for your GitHub setup.

Repository Overview

```
.
├── docs                        # Documentation and notes for the project / flow
│   └── info.md                 # Additional information about fields / flow
```

```
├── info.yaml                # Main project metadata (title, author, top_module,
source_files, pinout, etc.)
├── LICENSE                  # License for this template repository
├── README.md                # High-level project / template description
├── src                      # User RTL sources
│   ├── config.json          # Flow configuration (normally do not edit)
│   └── project.v            # Example user design (replace or add your own
modules here)
└── test                     # Simulation and cocotb testbench
    ├── Makefile             # Entry point for running simulations (`make -B`)
    ├── README.md            # Extra notes for the test environment
    ├── requirements.txt     # Python dependencies for cocotb test environment
    ├── tb.gtkw              # GTKWave view configuration
    ├── tb.v                 # Verilog testbench that instantiates your top-level
module
    ├── test.py              # Python cocotb testbench that drives tb.v and the
DUT
    ├── results.xml          # Test results from cocotb/pytest
```

---

3. Configure the Docker Development Environment

The project is designed to run inside a pre-built Docker image so everyone has the same tool versions.

3.1 Pull the correct Docker image

- **On amd64 (most desktop / laptop CPUs):**

```
docker pull jeshragh/ece183-293-win
```

- **On arm64 (e.g., Apple Silicon, some ARM boards):**

```
docker pull jeshragh/ece183-293
```

You only need to pull the image once (or when it gets updated).

3.2 Start a development container

From the **root of your cloned repo** (where `info.yaml` lives), run:

```
# taking amd64 (win/linux) device as an example
docker run --rm -it -v "$(pwd)":/workspace jeshragh/ece183-293-win bash
```

- `--rm`: Remove the container when you exit.
- `-it`: Interactive terminal.

- `-v "$(pwd)":/workspace`: Mount your current directory into `/workspace` inside the container.

> **Note:** On arm64, replace `jeshragh/ece183-293-win` with `jeshragh/ece183-293`:

```
# taking amd64 (win/linux) device as an example
docker run --rm -it -v "$(pwd)":/workspace jeshragh/ece183-293 bash
```

---

4. Apply Template Fixes (Compatibility Adjustments)

Depending on the original version of the template, you may need the following small fixes so that cocotb and the Makefile work as expected in this environment.

1. **Fix the cocotb clock parameter in `test/test.py`**
   - Change `unit` to `units` at line 13:

     ```
     clock = Clock(dut.clk, 10, units="us")
     ```

2. **Fix the Makefile variable name in `test/Makefile`**
   - In newer cocotb versions, the variable changed from `MODULE` to `COCOTB_TEST_MODULES`.
   - Around line 43 under `COCOTB_TEST_MODULES = test`, add:

     ```
     COCOTB_TEST_MODULES = test
     MODULE ?= $(COCOTB_TEST_MODULES)
     ```

Make sure these changes are saved in your repository (outside or inside Docker, but committed from the host).

---

5. Run the Example Simulation (Smoke Test)

With the Docker container running and your code mounted at `/workspace`:

```
cd /workspace/test
make -B
```

- `make -B` forces a rebuild and rerun of the simulation.
- This will simulate the example design (e.g., `counter.v` / `project.v` depending on the template version).
- If everything is set up correctly, the test should **pass** and produce results (e.g., `results.xml`, waveforms in `tb.fst`, and a `sim_build/` directory).

If the smoke test fails, check:

- That you used the correct Docker image for your architecture.
- That the `Clock` call and `Makefile` changes from Section 5 are correctly applied.

---

6. Create and Integrate Your Own Design

Once the example simulation passes, you are ready to add your own design.

7.1 Add your Verilog module(s)

1. Place your Verilog file(s) in `src/`, for example:
    - `src/counter.v` (already present as an example)
    - or your own file, e.g., `src/my_project.v`
2. Ensure your top-level module name is clearly defined (you will reference it in `info.yaml` and `tb.v`).

6.2 Update the Verilog testbench `test/tb.v`

- Edit `test/tb.v` so that it:
    - **Instantiates your top-level module** (from `src/`), and
    - **Connects the ports** (clocks, resets, I/Os) correctly.
- Align the signal names and directions with how the cocotb `test.py` expects to drive and observe them.

If you change signal names or add new I/Os, update both:

- The **Verilog testbench** (`tb.v`), and
- The **Python testbench** (`test.py`) accordingly.

6.3 Fill in `info.yaml`

Open `info.yaml` in the repository root and update the required fields:

- **`title`**: Short, descriptive name for your project.
- **`author`**: Your name and optionally contact.
- **`description`**: 1–3 sentence summary of what your design does.
- **`top_module`**: The exact name of your top-level Verilog module.
- **`source_files`**: List of all Verilog source files needed to build your design (e.g., `["src/counter.v"]`).
- **`pinout`**: Mapping of logical signals to physical TinyTapeout pins.

For the **pinout**, you have a lot of freedom in naming. Just make sure:

- The names match your top-level module's ports.
- You respect any TinyTapeout pin naming / width conventions required by the flow.

Consult `docs/info.md` and any course notes for the exact field semantics if needed.

---

7. Test Your Custom Design

After integrating your own design and updating `info.yaml`, re-run the simulation inside Docker:

```
cd /workspace/test
make -B
```

Check the test output:

- If the test **passes**, your design at least meets the basic functional checks encoded in `test.py`.
- If it fails, use:
  - The console log output,
  - The waveform file `tb.fst` (viewable with GTKWave, using `tb.gtkw`),
  - And assertions/messages in `test.py` to debug and iterate.

---

8. Commit and Push for Full CI Checks

Basic simulation is only the first step. To be **fabrication-ready**, you must pass the full set of GitHub Actions checks (including synthesis and layout).

1. **Exit the Docker container** (back to your host shell):

   ```
   exit
   ```

2. **Commit and push your changes**:

   ```
   git status          # optional, review changes
   git add .
   git commit -m "Implement my custom TinyTapeout design"
   git push
   ```

3. **Check GitHub Actions**

   - Go to your repository page on GitHub.
   - Click the **"Actions"** tab.
   - Open the latest workflow run triggered by your push.
   - Confirm that all relevant jobs have **passed**, typically including:
     - `gds` (physical layout / GDS generation)
     - `docs` (documentation generation)
     - `test` (simulation / verification)

If any job fails, click into it to view logs, fix the root cause locally, and push again.