

Digital ASIC Design Flow Overview using OSIC tools

Kithmin Wickremasinghe and Lohan Attapattu

April 27, 2025

Background of Open-source IC tools

The US CHIPS Act has allocated US\$53 billion to this industry, most of which has been earmarked for chip fabrication [1]. The question is, who will design the chips that must fill the capacity of these fabrication lines? There is a shortage of chip design talent in the USA and around the globe who have practical experience [2]. Gaining this experience takes years of training in a lab or industry setting. According to Peter Kinget, “providing practical experience is crucial in order to teach IC design successfully” [3]. Figure 1 illustrates the development stages of the industry chip design flow. OSIC tools and PDKs are a means for anyone in the chip design community to gain this experience themselves.

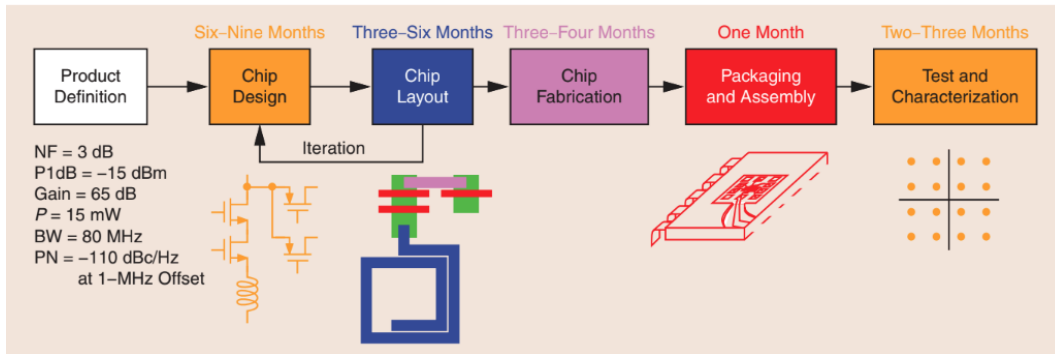


Figure 1: A typical chip development time frame [1].

Open-source Digital IC Design Flow

IIC-OSIC-TOOLS is an all-in-one Docker container for open-source-based integrated circuit designs for analog and digital circuit flows. There are numerous ways in which you can install them. The simplest, “default” method is to use the pre-packaged Docker container provided by: [IIC-OSIC tools](#). A research team led by Prof. Harald Pretl from Johannes Kepler University (JKU) developed this Docker container. Using this container, we can easily setup open-source tools and access it through Virtual Network Computing (VNC). You can find clear [template installation instructions](#) in SG13G_ASIC-Design-Template and also [installation instructions](#) on how to set this up on Dr. Kwantae Kim’s Blog. Regardless of how you install the tools, you will occasionally need to look at documentation. Here is a list of useful links:

- Xschem: http://repo.hu/projects/xschem/xschem_man/xschem_man.html
- KLayout: <https://www.klayout.de/>
- OpenROAD: <https://openroad.readthedocs.io/en/latest/>
- OpenLane: <https://openlane2.readthedocs.io/en/latest/index.html>

The digital design flow, also known as the ASIC (Application-Specific Integrated Circuit) design flow, is a series of steps that an engineer follows to design and analyze a digital circuit. Here’s a high-level overview of the process:

- **System Specification:** The design process begins with a system specification, which defines the functionality, performance, physical, and electrical parameters of the circuit.
- **Architectural Design:** The system specification is converted into a detailed architecture. This includes defining the data path, control units, memory blocks, and other high-level components.
- **RTL Coding:** The architecture is then coded in a Hardware Description Language (HDL) such as Verilog or VHDL at the Register Transfer Level (RTL). This describes the flow of signals and the operation of the data path.
- **RTL Verification:** The RTL code is verified using simulation tools to ensure it meets the original specifications.
- **Synthesis:** The RTL code is converted into a gate-level representation using a synthesis tool. This process maps the RTL code to a specific technology library, which contains the logic gates and flip-flops used in the design.
- **Gate-Level Verification:** The gate-level netlist is verified using simulation tools to ensure it still meets the original specifications.
- **Place and Route (P&R):** The gate-level netlist is placed onto a chip and the interconnections are routed. This process also includes clock tree synthesis and power grid design.
- **Physical Verification:** The layout is checked for any violations of the design rules of the fabrication process. This includes Design Rule Checking (DRC) and Layout Versus Schematic (LVS) checks.
- **Timing Analysis:** The design is checked to ensure that it meets the required timing constraints. This includes setup and hold time verification, as well as checking for any timing violations.
- **Tape-out:** Once the design has passed all checks, it is sent for manufacturing. This process is known as tape-out.

OpenLane is an automated, open-source digital design flow that facilitates the process of designing integrated circuits (ICs). It is a part of the OpenROAD project [4] and is designed to carry out all ASIC implementation steps, from RTL (Register Transfer Level) all the way down to GDSII (Graphic Data System version II), a standard for data exchange of IC layout artwork.

OpenLane integrates several open-source tools such as OpenROAD, Yosys, Magic, and Netgen, along with custom methodology scripts for design exploration and optimization. This integration abstracts the various steps of silicon implementation, enabling users to harden their designs using simple configuration files.

There are two versions of OpenLane: OpenLane 1, a stable silicon implementation platform built entirely on open-source software, and OpenLane 2, a next-generation silicon implementation platform for highly-custom chips. OpenLane 1 offers a simple, Docker-based installation and has enabled countless tape-outs for Efabless’s chipIgnite and Google’s Open MPW programs. On the other hand, OpenLane 2 allows users to write fully custom ASIC implementation steps or flows.

OpenLane is gaining popularity in the field of Electronic Design Automation (EDA) for its stability, flexibility, enhanced user experience, and customizability. It is particularly useful for researchers, students, and small-scale projects due to its cost-effectiveness and transparency. The digital design flow using OpenLane EDA tools involves several stages as shown in Fig. 2, each of which utilizes different open-source tools integrated within OpenLane. Here’s a high-level overview of the process:

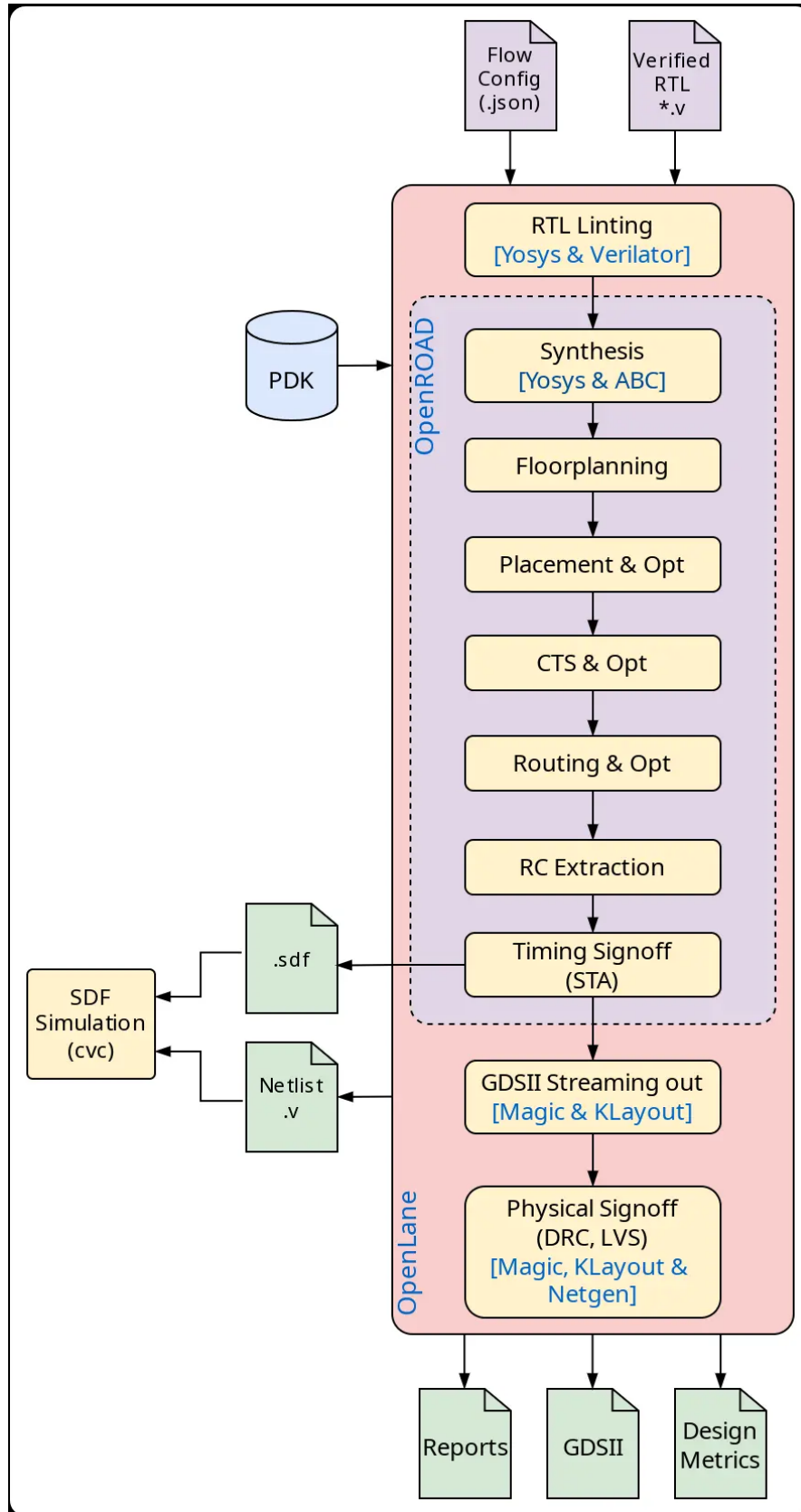


Figure 2: The open source digital IC design flow using OpenLane EDA tools [5].

- **RTL Synthesis:** The design flow starts with RTL (Register Transfer Level) description in Verilog. This is synthesized to gate-level representations using the Yosys tool.
- **Technology Mapping:** The synthesized output is then technology-mapped using the ABC tool.
- **Floorplanning:** The floorplan of the design is created, which includes the core area, locations of pre-placed cells, and the power structure.
- **Placement:** The design's standard cells are placed in the rows created during floorplanning. This is done using the OpenROAD app.
- **Clock Tree Synthesis (CTS):** A balanced clock tree is synthesized for the design to minimize clock skew across sequential elements. This is done using the TritonCTS tool.
- **Routing:** The placed design is routed using the FastRoute tool for global routing followed by detailed routing using the OpenROAD app.
- **Signoff:** Finally, the design is checked for any violations and corrected. This includes Design Rule Checking (DRC) using the Magic tool, Layout Versus Schematic (LVS) check using the Netgen tool, and timing analysis using the OpenSTA tool.

This flow is iterative, meaning that if the design fails at any stage, it is optimized and the flow is run again. OpenLane automates this entire process, making it easier for designers to go from RTL to GDSII with less manual intervention.

Note: Efabless has shut down operations in March 2025 due to funding challenges. This includes Efabless's OpenLane project. Hence, we will stick to using only OpenROAD in this tutorial.

IHP SG13G2 PDK

SG13G2 is a high performance BiCMOS technology with a 130nm CMOS process: [IHP 130nm BiCMOS Open Source PDK documentation](#). It contains bipolar devices based on SiGe:C npn-HBT's with up to 300 GHz transit frequency (fT) and 500 GHz maximum oscillation frequency: [BJT Devices](#). The SG13G2 process provides 2 gate oxides: A thin gate oxide (sg13.lv) for the 1.2 V digital logic and a thick oxide (sg13.hv) for a 3.3 V supply voltage: [MOSFET Devices](#). For both modules NMOS, PMOS and isolated NMOS (iNMOS) transistors are offered. The backend option offers 5 thin Al metal layers, two thick Al metal layers (2 and 3 μm thick) and a MIM layer. The metals are organized as Metal1-Via1-Metal2-Via2-Metal3-Via3-Metal4-Via4-Metal5-MIM formation-TopVia1-TopMetal1-TopVia2-TopMetal2-Passivation.

IHP-Open-DesignLib is repository, which contains open source IC designs using IHP SG13G2 BiCMOS process: [IHP-Open-DesignLib documentation](#). It is also a central point for design fabrication under the concept of IHP Free MPW runs funded by a public German project FMD-QNC (16ME083). Project funds can be used exclusively to produce chip designs for non-commercial activities, such as university education, research projects, and others. In the project, a continuation for the provision of free area for the open source community is to be worked out. The criteria for the selection of designs submitted by the open source community are specified here. In order to set IHP-SG13G2 as the default PDK in xschem, run the command `iic-pdk ihp-sg13g2`.

Once the projects have passed physical verification (DRC/LVS) and post-layout simulations after parasitic extraction (PEX), then the [filler generation process](#) is done using KLayout. The maximum area granted to a community member is 2 mm^2 , which includes the [sealring](#). An example of the area provided for the design is given in the [test.layout.gds](#) file, which is a 300 μm \times 300 μm space. After the final GDSII file has been generated and submitted, the selected designs will be processed at the IHP pilot line facility (clean room). This process takes around 4 to 6 months depending on the technology. After this process, the designers can obtain the chip die and package the die into QFN32 chips.

Flow Demonstration

This section is mainly based on the [SG13G2_ASIC-Design-Template](#), which is suited for analog mixed-signal design with the 130nm BiCMOS open-source SG13G2 PDK by IHP and the IIC-OSIC-Tools by the Institute for Integrated Circuits and Quantum Computing (IICQC), Johannes Kepler University, Linz (JKU).

1. Step 01 - Using SSH to clone GitHub Repositories

- If you are not familiar with GitHub cloning using SSH, then watch this [video](#).

2. Step 02 - Install Docker and IIC-OOC toolkit





- Follow the instructions in this [link](#) and install the docker with X11 forwarding. If any issue arises, then go through the README file of this GitHub [repo](#). Make sure to cover the [display](#) and [design-path](#) steps.

Do the following as shown in the above video:

- Open `start_vnc.sh` file
- Search `PARAMS`
- Assign `${PARAMS} -e DISPLAY=host.docker.internal:0` to `PARAMS`

3. Step 03 - Design folder path selection (Already completed in Step 02)

- In the above procedure, you have selected a design folder as the local folder for your Docker project environment, and all the steps below should be executed inside this "designs" folder.

- Assign your working directory to `DESIGNS`
 - In  Windows, your path `Documents/open-source` in  Windows is automatically mounted to `/mnt/c/Users/{your_username}/Documents/open-source` in  WSL. For simplicity, I recommend to use the path under `Documents` folder
 - In  MacOS, any path you want can be used
- Save `start_vnc.sh`

4. Step 04 - Cloning the GitHub repository

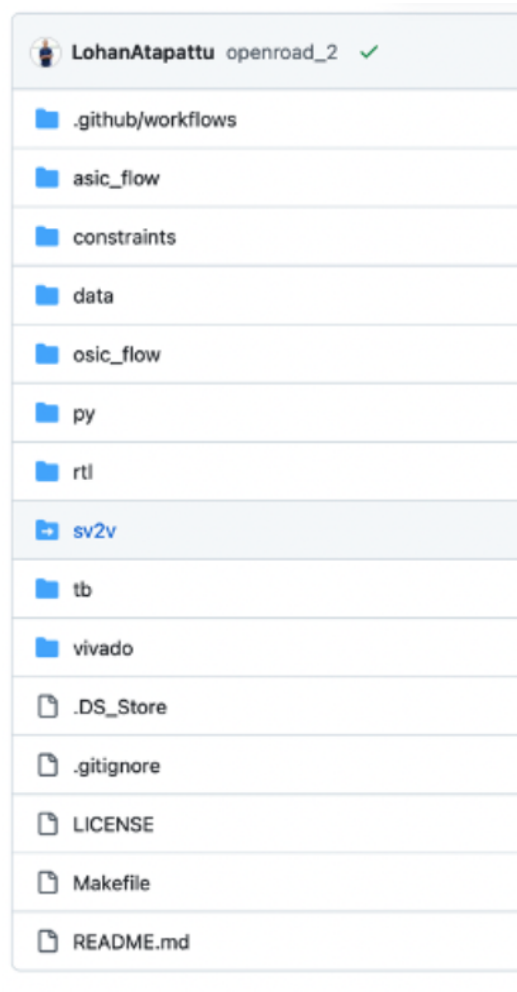
- Go to the [branch](#) of main/osic_flow_openroad of SkillSurf GitHub repo.
- To clone a repository to your "designs" folder, use the command
- `>> git clone git@github.com:SkillSurf/systemverilog.git`
- Clone the GitHub repository to the "designs" folder you created

```
/foss/designs > git clone git@github.com:SkillSurf/systemverilog.git
```

5. Step 05 - Build the SV2V project

- You need to convert all SystemVerilog (.sv) files to Verilog (.v) in order to use OpenROAD. To do this we use "sv2v" folder in the above cloned repository
- Use command line interface (CLI) or terminal to execute the below commands (do not use Docker terminal)

- Inside "sv2v" folder, build the following project from source based on the following GitHub [repo](#).



Building from source

You must have [Stack](#) installed to build sv2v. Then you can:

```
git clone https://github.com/zachjs/sv2v.git
cd sv2v
make
```

This creates the executable at `./bin/sv2v`. Stack takes care of installing exact (compatible) versions of the compiler and sv2v's build dependencies.

You can install the binary to your local bin path (typically `~/.local/bin`) by running `stack install`, or copy over the executable manually.

6. Step 06 - Convert SV2V

- Open your terminal and navigate to the directory containing your .sv files
- Then use the following commands to convert files; i.e. follow these steps in your computer terminal or command prompt (Do not use Docker terminal)

7. Step 07 - IC layout generation

```
sh

cd /path/to/your/systemverilog/files
```

1 Convert .sv to .v (Same Directory)

If you want to create .v files next to each .sv file:

```
sh

sv2v --write=adjacent fpga_module.sv
```

This generates fpga_module.v in the same directory.

2 Convert .sv to a Specific .v File

If you want to convert fpga_module.sv to a specific Verilog file:

```
sh

sv2v --write=fpga_module.v fpga_module.sv
```

Now, fpga_module.v will contain the converted Verilog.

3 Convert Multiple .sv Files to a Directory

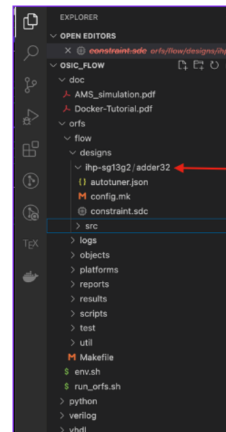
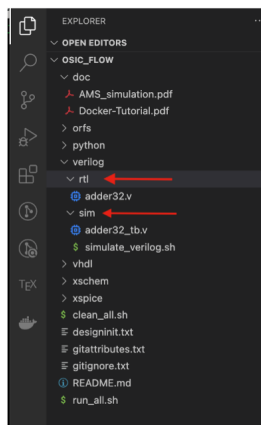
If you have multiple .sv files and want them in a separate directory:

```
sh

mkdir converted_verilog
sv2v --write=converted_verilog *.sv
```

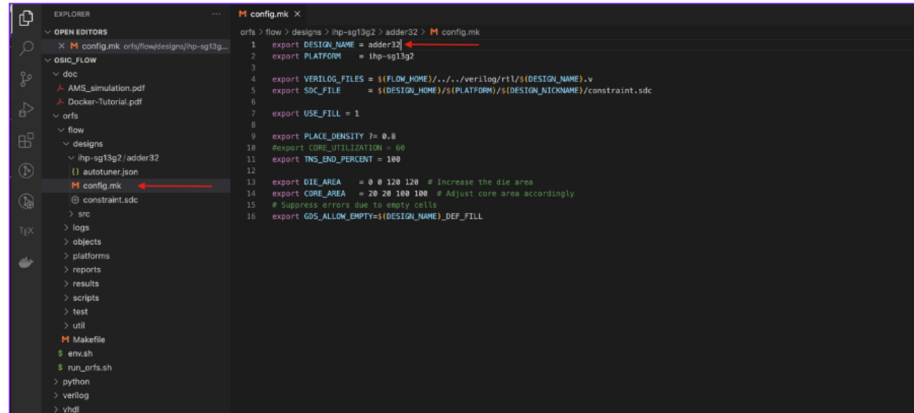
This puts all converted .v files into converted_verilog/.

- Copy the .designninit file from "foss/designs/SG13G2_ASIC-Design-Template" to "foss/designs" and check the path for Xschem in the file. This will change the default PDK to IHP SG13G2.
- Navigate to "osic.flow" directory through VSCode or manually

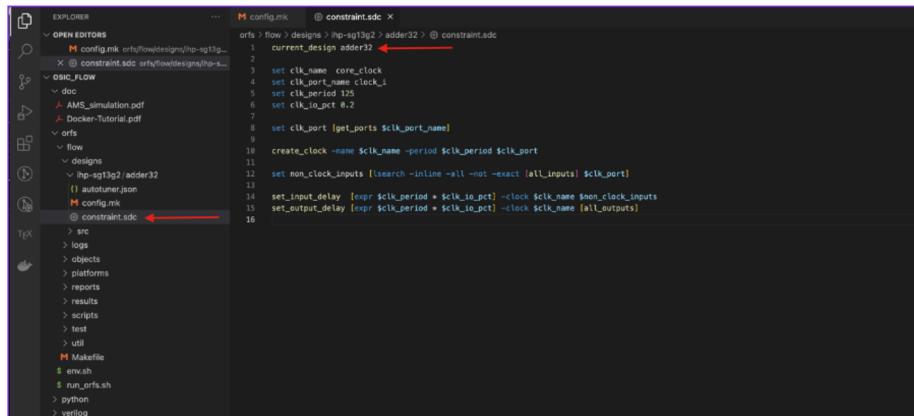


- Navigate to "verilog" folder.
- Add the Verilog (.v) source files to "rtl" folder and simulation files to "sim" folder.

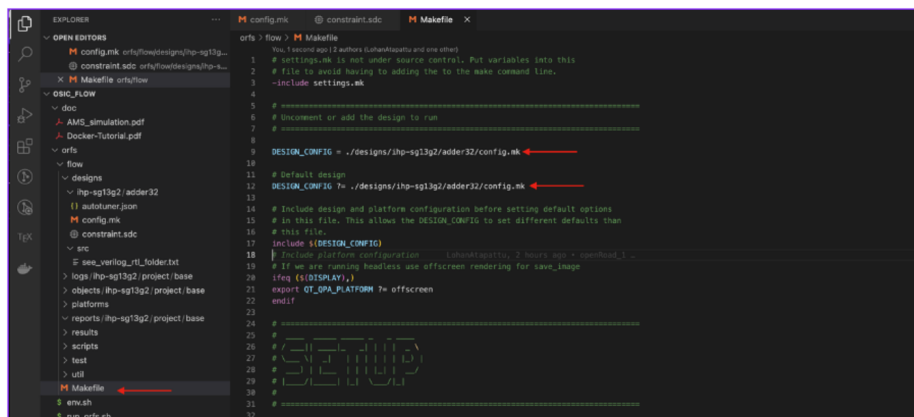
- Change the name of the folder according to your design name (eg: adder32).
- Change the name of the configuration file (config.mk) DESIGN_NAME



- Change the current_design name of constraint.sdc file. You can read more about the flow variables [here](#).



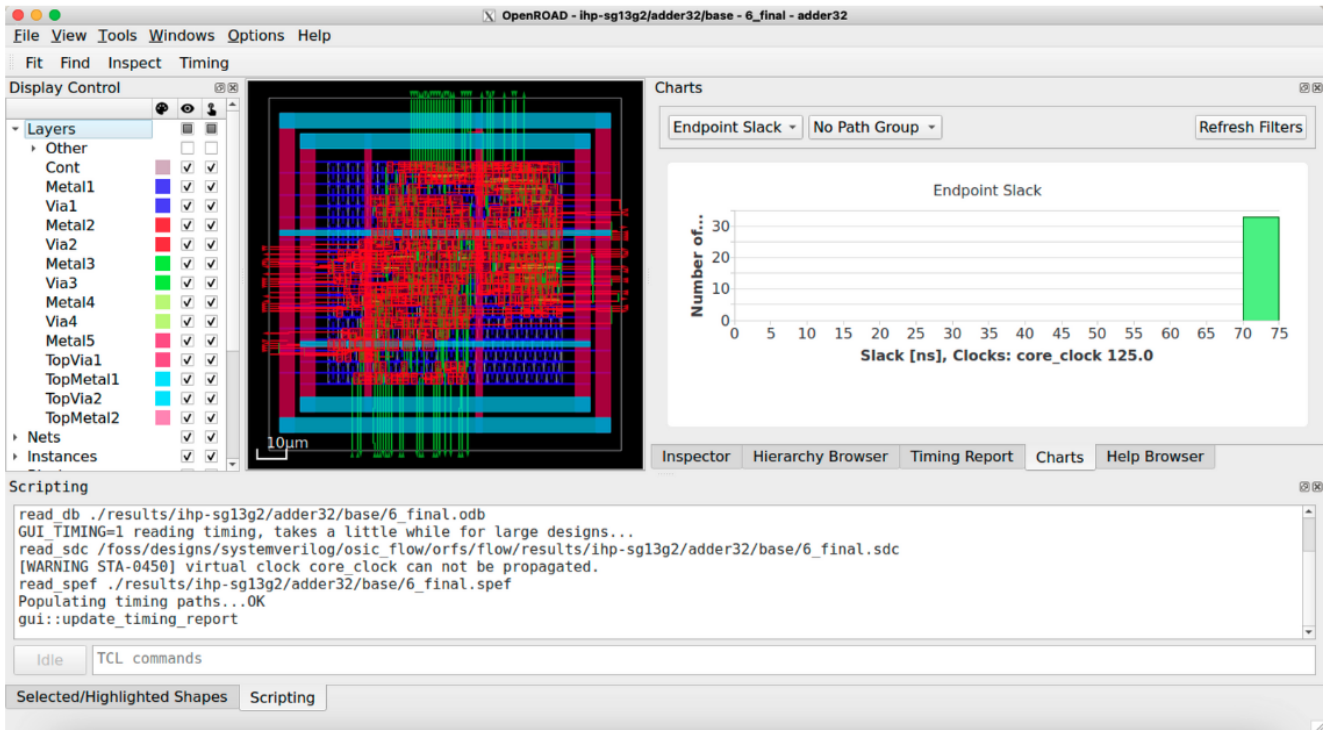
- Change path in Makefile as shown below.



- Through the Docker terminal, go to the osic_flow directory and run the following command:

```
/foss/designs/systemverilog/osic_flow > ./run_all.sh
```

- This will finish your design as shown in the OpenROAD GUI.



Project Description - The design of a digital stopwatch

Introduction

The general architecture of a state machine circuit is shown in Fig. 4. The figure shows the state register, and two combinational logic circuits – the next state logic and the output decoder blocks. The next state logic circuit is responsible for driving the D inputs of the flip-flops in the state register. Next state logic inputs include the present state (which is the output of the state register), and the overall circuit inputs. This information is combined in the next state logic circuit so that next-states can be based on the present state, and on overall circuit inputs. The next state logic circuit can be designed using the techniques presented earlier – namely, a truth table to show the input-output relationships, and K-maps or computer algorithms to find optimal circuits. But, when truth tables are used like this (to define logic circuits that are in the feedback path of registers), it is difficult to see the “bigger picture”, or the sequence of states that are implemented by the next state logic circuit. A better design method uses a state diagram to capture the requirements – state diagrams are discussed in the background topics.

The output decoder circuit receives the present state signals as inputs, and creates useful output signals based on the present state. In the example presented in the opening paragraph above, the present state signals might indicate that the traffic light controller state machine is in the “turn on the yellow light” state. That state would be associated with a unique binary number in the state register, and the output decoder would receive that state code, and from it, create a signal that could turn on the yellow light. The output decoder circuit requirements could also be captured in a truth table, and then a circuit could be designed based on that table. But, as with the next state logic circuit, the use of a state diagram makes it easier to visualize how output signals are generated from various states.

State machines, like other circuits we’ve encountered, can be designed using structural methods or behavioural Verilog. Both structural and behavioural design methods are presented in the background topics, but behavioural Verilog is by far the preferred method. We will use behavioural Verilog in this project. Behavioural Verilog for state machines can be created following many different models. Which

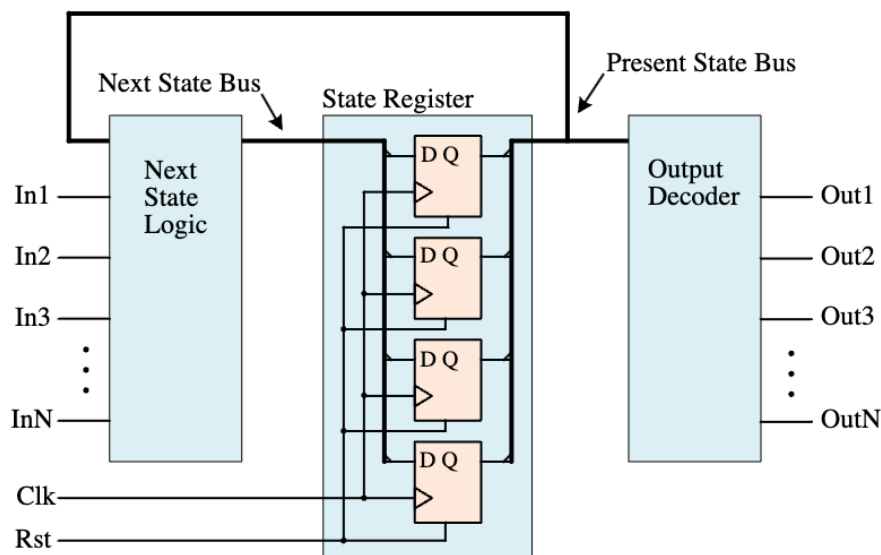


Figure 4: General model of sequential circuit.

model you use will depend on your personal experience and stylistic preferences. In the tutorial below, we present a model that uses two Verilog always blocks: one to describe the state register, and a second to describe the next-state combinational logic. The state-register block is no different from the code used earlier to describe Verilog registers. The next state block uses a case statement to make the code more readable. Note this is a stylistic choice – for all but the simplest state machines, clearly showing the next-state logic functions to the code reviewer/reader is a meaningful goal, and a case statement helps in that regard. If you use an always block (and you should strongly consider it!), be sure to cover all possible combinations of the inputs in the sensitivity list to avoid creating unwanted latches.

Stopwatch with Start, Stop, Increment, and Clear Functionality

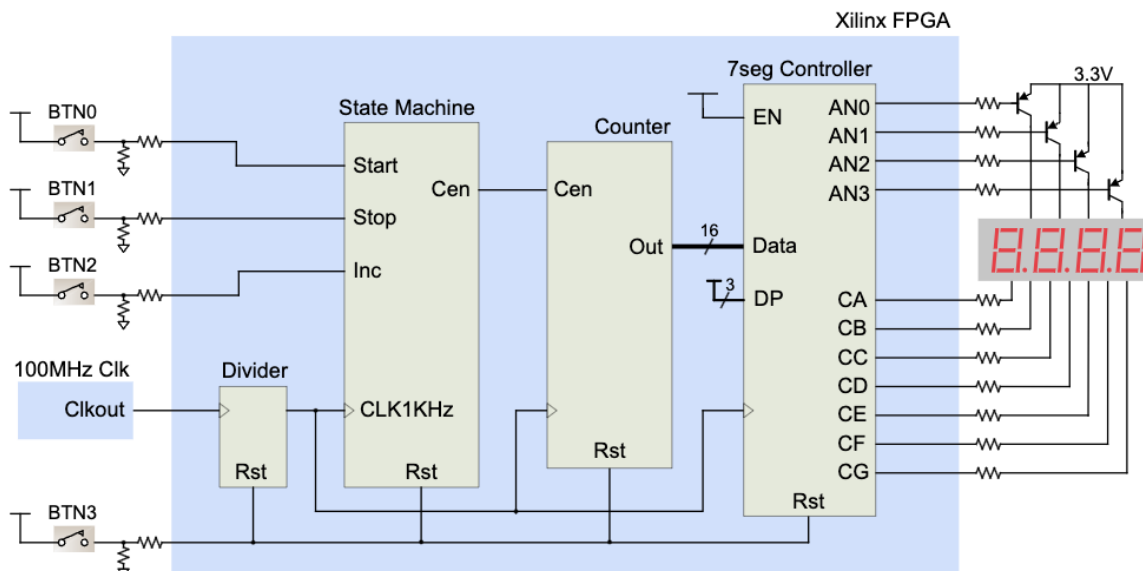


Figure 5: Stopwatch block diagram.

Hint #1 - System block Diagram

A block diagram for the stopwatch is shown in Fig. 5. Note that not all behaviors are fully specified in the design requirements section above. For example, if the timer is currently running and the increment button is pressed, no requirement is given to specify how the timer should behave. As an engineer, you must understand the design intent and create behaviours for underspecified behaviours that are consistent with your understanding of the intent. In practice, it is difficult (or even impossible) to completely specify every behaviour in a written document, so engineers frequently invent behaviours where specifications are lacking. Figures providing greater detail for the blocks shown above are shown below.

Hint #2 - Decimal Counter

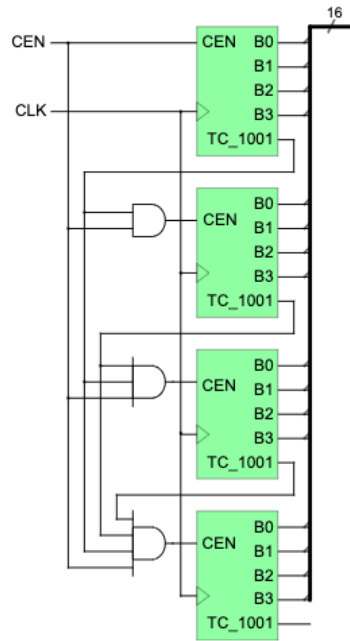


Figure 6: 4-digit Decimal Counter.

The timer requires a four-digit decimal counter as shown in Fig. 6. Such a counter can be built from four individual counters that each count between 0 and 9 (0000 to 1001), and then roll over from 9 back to 0, creating a continuous count pattern with 10 binary numbers.

Individual counters are based on 4-bit binary counters that each detect when the pattern 1001 is present on their outputs. When 1001 is detected, the counters reset themselves back to 0000 and issue a terminal count (TC) signal to indicate the count range is complete and the counter is resetting back to zero. Each counter also uses an enable signal (CEN). When CEN is asserted, the counter will increment with each clock edge, but when it is deasserted, the counter will ignore the clock and hold at its present state. The CEN signal can be used to enable more significant counters each time a less significant counter completes its count range.

The TC signal is typically driven by a logic gate combining the count bits according to some function (in our case, TC_1001 is generated by a 4-input AND gate detecting 1001). TC_1001 will be asserted for as long the 1001 pattern is present on the counter outputs. In the least significant counter, this is a single clock period. But in the next most significant counter (representing the 10ms position in a four digit decimal number), the 1001 pattern will be present for 10 clock periods, and so its TC_1001 signal will be asserted for 10 system clock periods. To keep the next most significant counter from counting 10 times (during the period TC_1001 is asserted), the TC_1001 signal from the first counter must be

combined with the TC_1001 signal from the next most significant counter. In fact, a little thinking about the problem will reveal that each TC_1001 from more significant counters must be combined with all such signals from all less significant counters. The outputs of the four individual counters can be assembled into a single 16-bit bus for transport to the seven-segment display controller.

Hint #3 - Seven Segment Display Controller

The block diagram in Fig. 7 shows a structural design of the seven-segment display controller.

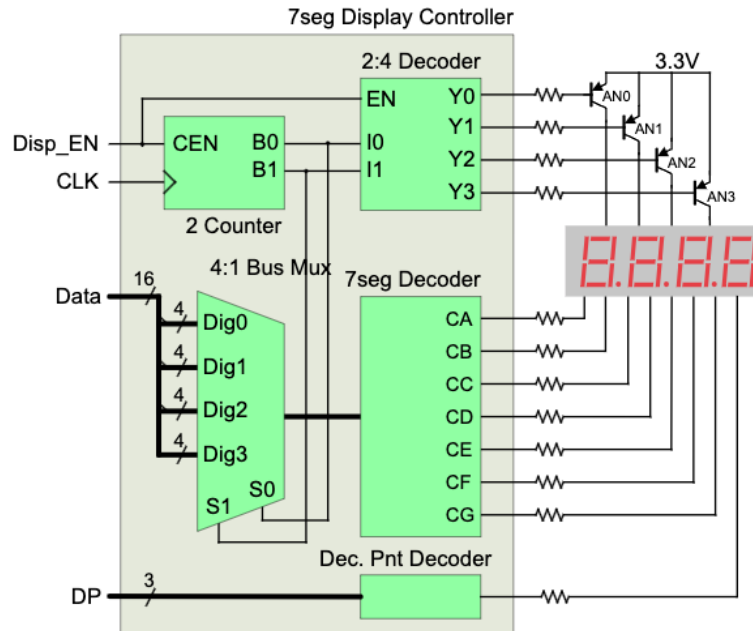


Figure 7: Seven Segment Display Controller.

For example, if the timer is currently running and the increment button is pressed, no requirement is given to specify how the timer should behave. As an engineer, you must understand the design intent and create behaviours for underspecified behaviours that are consistent with your understanding of the intent. In practice, it is difficult (or even impossible) to completely specify every behaviour in a written document, so engineers frequently invent behaviours where specifications are lacking.

Design Requirements

1. SystemVerilog Architecture: Create a four-digit stopwatch module that uses a seven-segment display as an output device. The stopwatch should count from 0.000 to 9.999 seconds and then roll over, with the count value updating exactly once per millisecond. The stopwatch uses three pushbutton inputs: start, stop, increment, and clear (reset). The start input causes the stopwatch to begin incrementing at a 1 KHz clock rate (i.e., one count per millisecond); the stop input stops the counter from incrementing but leaves the display showing the current counter value; the increment input causes the displayed value to increment once each time the button is pressed regardless of how long the increment button is held down; and the reset/clear input forces the counter value to zero. An example project is available [here](#).
2. Visualize Waveforms: Use [EDA playground](#) or [makerchip](#) to verify the design by visualizing the waveforms of each of the implemented modules.

3. Demonstrate ASIC flow: You have to take the above digital stopwatch design Verilog files, and synthesize the RTL using the IHP SG13G2 PDK and OSIC tools. Next, you must get the final layout after doing PnR using the OpenROAD. Finally, you must export the final .gds file and visualize it using kLayout <https://www.klayout.de/build.html>. As the final step of the ASIC flow, you can further follow a project similar to [301 Simple Stopwatch](#) in order to tapeout your design in TTIHP25b.
4. FPGA Implementation (Optional): Implement the design on any FPGA board and demonstrate the functionality similar to the following [video](#).

Optional: Assignment 3 - ASIC flow with OSIC Tools

The verilog files are available [here](#). You can find this project submitted for tapeout in TTIHP25a [here](#).

References

- [1] B. Razavi, “Education of chip designers at a large scale: A proposal [society news],” *IEEE Solid-State Circuits Magazine*, vol. 16, no. 2, pp. 76–83, 2024.
- [2] The Economist, “America is building chip factories. now to find the workers,” *The Economist*. [Online]. Available: <https://www.economist.com/united-states/2023/08/05/america-is-building-chip-factories-now-to-find-the-workers>
- [3] P. Kinget, “Teaching ic design: From concepts to testing a fabricated custom chip [society news],” *IEEE Solid-State Circuits Magazine*, vol. 15, no. 3, pp. 87–93, 2023.
- [4] T. Ajayi, M. Neseem, G. Pradipta, S. Reda, M. Saligane, S. Sapatnekar, C. Sechen, M. Shalan, W. Swartz, L. Wang, Z. Wang, V. Chhabria, M. Woo, B. Xu, M. Fogaça, S. Hashemi, A. Hosny, A. Kahng, M. Kim, and U. Mallappa, “Toward an open-source digital flow: First learnings from the openroad project,” 06 2019, pp. 1–4.
- [5] M. Shalan and T. Edwards, “Building openlane: A 130nm openroad-based tapeout- proven flow : Invited paper,” in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–6.