SECTION OF ELECTRICAL ENGINEERING

EPFL STI – SEL
ELG 030
Station nº 11
CH-1015 Lausanne

Téléphone : +4121 693 13 46
Fax :
E-mail : alexandre.levisse@epfl.ch
Site web : https://sti.epfl.ch/fr/sel/

Fundamentals of VLSI

September 2025

FVLSI - Digital Design Flow
Front End – from RTL to Synthesis , Preamble

1. Objectives

The goal of this class is to teach you the basics of the digital design implementation flow and tools that are conventionally used in the industry. You will be introduced to the methodology allowing you to translate HDL code into ready-for-fabrication circuits.

This second phase of the lab will, as the first phase, focus on a 2-step "learn by practice" approach. First you will learn by an example following a tutorial. Then, you will practice your newly acquired skills through mini-projects.

2. Introduction

2.1. (Simplified) Digital and verification design flow

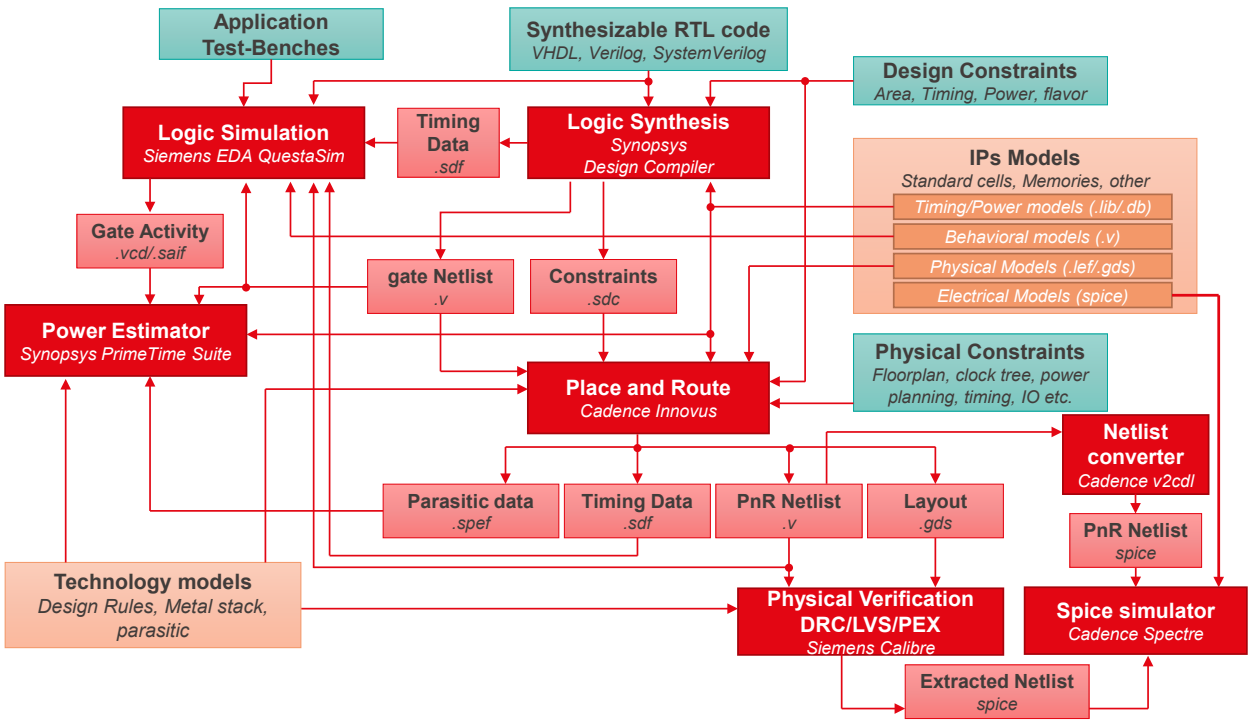

Figure 1: Schematic diagram of the Digital design flow used in this series of Labs

The design flow presented in Figure 1 summarizes the knowledge you will have at the end of this series of 5 labs dedicated to the digital design flow. This schematic diagram represents :

- With Green boxes, the designer inputs
- With Red boxes, the software EDA tools used for each step
- With Orange boxes, the technology inputs (from the foundry) and Intellectual Property (IP) blocks (from the foundry or other design compagnies)
- With light Red boxes, the files being used as each step.

The next sections present the flow through each EDA tool being used. Note that the logic simulation step will be described several times, as along the flow, logic simulation has to be run to verify the circuit functionality at different stages of the design.

- *This design flow schematic diagram is simplified and does not cover for many more design and verification steps which could be inserted. Typically, we do not show Automatic Test Pattern Generation (ATPG) and scanchain insertion which can be done on the gate netlist. This step is mandatory in industrial designs as it allows the providers to test the chips. We also do not show topographical synthesis, equivalence checking, IRdrop evaluation, Static Timinc Analysis (STA), Design for Manufacturing (DFM)[1] etc[2].*
- *This lab also focuses on the design of a circuit block instantiating memory IPs and standard cells. When going for sign-off (ready for fabrication), there are a few more steps which we do not describe here, and that we could propose that you explore during a semester project with one of the labs working on the topic. These steps are for instance, pad ring design or dummy filling.*

2.2. LOGIC SIMULATION – RTL

Logic simulation is the first step to perform when starting a design. It is used to verify the functionality of a RTL code written by the designer. Two inputs are needed by the simulator at this point : the RTL code to be tested, and a testbench.

At this point in the design, no timing checks are being performed. A functional simulation at the RTL stage DOES NOT ensure functionality of the circuit.

2.2.1. SYNTHESIZABLE RTL CODE : THE CODE THAT REPRESENTS THE CIRCUIT TO BE DESIGNED

You will hear a lot about what is the best RTL language to use, and many different motivations/reasons. It is true that Verilog or SystemVerilog are the leading languages today.

[1] https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/dfm-overview.html
[2] https://www.cadence.com/en_US/home/tools/digital-design-and-signoff.html
https://www.synopsys.com/implementation-and-signoff/fusion-design-platform.html

Keep in mind that the end language is not that a big deal. The knowledge acquired when learning RTL modeling is not about the language, it is about how to properly and correctly describe hardware. Generally, in EPFL we make the choice of teaching you VHDL. Switching to (system)Verilog can be done easily at any point in your career. What matters is that you do not write your RTL code as you would write c-code, but put yourself in the context of the design of a hardware block. Learning Verilog without that background can sometimes lead to confusion as the syntax is much more permissive and less verbose than VHDL. By forcing you to properly define in a verbose way, the different part of your code, VHDL gives you go design practice that will help you later on.

RTL[3] (Register Transfer Level) models *describe the structure and the behavior of the design at a relatively high level of abstraction and provide a clear separation between control parts (e.g. finite state machines - FSM) and operative parts (e.g. arithmetic and logic units). Registers are used to store small size data between clock cycles.*

RTL can be represented with several languages. The most used ones are VHDL, Verilog or System Verilog. RTL code can also be automatically generated from C code using High-Level Synthesis (HLS[4]) through dedicated tools.

2.2.2. APPLICATION TESTBENCHES :THE CODE THAT REPRESENTS THE BEHAVIOR TO BE APPLIED TO THE RTL DESIGN.

A testbench can be written by hand in Verilog/VHDL. Or could be automatically generated by various toolchains. For e.g. python based toolchains exist and could be of interest in some cases[5].

What matters when you write a testbench (remember here what you did in the first phase of these labs, when designing your adder) is that it does properly test the functionality of your circuit and covers potential errors that you want to spot during the simulation part.

The testbench will also be used later for power estimation. The power consumption of a circuit varies with its activity. Operations/applications that trigger high power consumption are most likely not the same as the ones you want to test when verifying the functionality of your circuit. When doing power evaluations, you may want to use different testbenches.

2.3.RTL/LOGIC SYNTHESIS

The RTL (or logic) synthesis step transforms the RTL code into a circuit composed of logic gates based on some constraints provided by the user. To achieve this end, the logic synthesis tool uses:

- Timing and power models from a standard cell library (remember the UMC65LL_UMK65LSCLLMVBBR__B03PB lib on virtuoso ?[6]). represented as a .lib or .db file. The standard cells are an ensemble of logic gates (AND, NAND, NOR, MUX,

[3] https://en.wikipedia.org/wiki/Register-transfer_level
[4] https://en.wikipedia.org/wiki/High-level_synthesis
[5] https://www.cocotb.org/
[6] This is an example of a standard cell library as provided by UMC for their 65nm CMOS technology.

INV, ADDERS etc.) which will be used by the synthesis tool to build a circuit. On top of the gate types, the standard cell also contain various sizes for the same gate (called *fanout* or *drive*) which are used during the timing analysis in order to meet the user-defined timing constraints.

- Design constraints provided by the user. These constraints are provided to the tool in the form of commands in the tcl language[7]. The set of available commands can be written to the tool in the corresponding command prompt, or chained in a script that can be called through the tool.

- *Tcl can be painful to use sometimes… but it is what it is for now… some open source initiative use python-based UI, though it is far from being the norm.*
- *Some synthesis tools also provide a Graphical User Interface (GUI) though, these are not really made to be used. There are several reasons for it. One is that the process of performing logic synthesis does not require to be visual, and that once the flow is stabilized, there is no much use for a GUI besides some visualizations. The second reason being that the design flows are complex and constantly evolving. Thereby the documentation of DesignCompiler contains several 1000s pages which are regularly updated from versions to versions. Reading such documentation is a daily duty of a digital designer engineer.*
- *You will find on moodle 3 documents*
    - *The Design Compiler User Guide Version 2022.12*
        - *This 879pages document presents the design compiler suite and proposes methodologies on how to use it, reference flows etc.*
    - *The Design Compiler Tool Invocation Command 2022.12*
        - *This document presents the syntax for the different terminal commands available as part of the design compiler suite.*
    - *The Synthesis Tool Commands Version 2022.12*
        - *This document is the most important one being used during this lab. It presents all the commands that can be used in design compiler (as well as inside the tcl scripts) with the detail of their syntax. Expectedly this document is extremely long (6100 pages). You are not expected to read it all, but should be able to navigate through it and search commands in there if there are things you do not understand or do not find clear.*
- *These documentations associated with EDA tools are generally made available by EDA providers on their web interface for their customers. A good practice is to request access to their support interface to your manager in your future job. As this will give you access to the latest and up to date documentation, trainings, example flows for the version of the tool you use.*

At the end of the process, the synthesis tool ultimately generates :

- A Verilog netlist[8] containing standard cells. Which we can simulate again to perform a post-synthesis logic simulation. This netlist does only contain a list of gates, which do not

[7] Tcl (Tool Command Language - https://en.wikipedia.org/wiki/Tcl), pronounced “tickle” is used by EDA tools providers as their default language.

[8] A VHDL gate-level netlist could be also generated. The main reason to use a Verilog one is that the place-and-route tool only understands this format as input. Actually, many EDA tools only understand Verilog inputs, especially when such inputs are supposed to be generated by other tools.

contain any timing information. This netlist will be used as an input for the place and route process later on.

- *The post synthesis netlist could also be used as an input for a ATPG and scan chain insertion (cf EE-530 class).*

- A delay file (Standard Delay File – SDF format[9]). The SDF description includes delay information for simulation. Note that in this lab, the flow we use does not consider parasitic information post Logic Synthesis. Thereby the only delay info being included in the sdf file at this point are the capacitive load of the gates, and their transmission time.
- A constraint file (Synopsys Design Constraints - SDC[10]). *Yes, it does contain the name Synopsys in it, but it is kind of a standard*. The SDC file is just a translation of the commands you did input during the synthesis process.

  - *Generally most of the commands are properly translated in the sdc file, though, if the tools used after synthesis are from a different provider, it can happen that some sdc commands are not fully understood and must be re-specified.*

It can be important to note here that recent tools provide pre-synthesis floorplan-aware synthesis. In other words, translation of the HDL code can be done knowing a preliminary floorplan of the chip. Though these flows can be extremely efficient (as the netlist is optimized knowing the future physical placement) and are almost required when designing circuits in advanced technologies (sub 10nm), they are specific to each provider and out of the scope of this lab.

In 2023, tools such as the genus-innovus flow (Cadence), fusioncompiler (Synopsys) or Aprisa Suite (Siemens) provide this feature. The interface between synthesis and Place and Route is handled through proprietary binary files.

2.4. POST-SYNTHESIS LOGIC SIMULATION

The post-synthesis gate-level simulation normally uses the same testbench models as the ones developed for the verification of RTL models and the same logic simulator. The simulation of the Verilog netlist requires models of the standard cells that are usually provided in the design kit. These models can be back-annotated with timing delays from the SDF file generated in synthesis. The post-synthesis simulation does not provide yet a completely accurate verification as only the timing delays of the physical cells are considered. The interconnect delays are however either ignored or at best estimated.

The post synthesis step is generally extremely important in the design flow, as it allows the designer to verify that their design was correctly encoded in RTL language. Generally, a wrongly written RTL leads to incorrect functionality post synthesis. Once this step passed, and the circuit functionality validated post synthesis, it is most likely that the circuit is correct.

[9] https://en.wikipedia.org/wiki/Standard_Delay_Format
[10] https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_sdc.htm

During a post synthesis simulation, the designer can extract, for a given testbench, a gate-activity file which contains the switching information of each gate in the circuit. This will then be used to extract power information. There are two types of formats that can be used :

- Value Change Dump (VCD[11]), which contains the switching activity of each single gate along time. This file can be extremely heavy, for large circuits, but is extremely precise.
- Switching Activity Exchange Format (SAIF[12]), which contains a statistical representation of the switching activity for a given period. It is more compact than VCD format, but does also contain less information. Typically, it does not allow the user to catch a "peak" activity.

2.4.1. STANDARD CELL PLACEMENT AND ROUTING

The *place-and-route* (PnR) step transforms the Verilog gate netlist generated by Logic Synthesis into a physically "fabricable" circuit in a format called GDSII[13]. This file is actually a translation of the physical layout, as you did practice in the first phase of this lab, but automatically generated by the PnR tool.

The PnR step consists in organizing physically the IP blocks (memories and other non-synthesizable blocks), design the power grid, placing the standard cells in rows of equal height and routing them together. It will also perform the clock tree insertion. Along these steps, the tool will perform some physical verification, timing analysis (setup and hold) and ensure that everything is connected where it is supposed to be. Though, as usual, the tool does only know what is given to it, and additional verification with tools qualified as "sign-off" is always better (typically, a DRC and LVS steps **HAVE TO BE** run on a post PnR design before sending it for fabrication).

The PnR tool (here we use Innovus) has access to :

- The standard cell library physical information to perform the placement. All the standard cells from a library have the same height, but may have different widths. Each cell has a power rail at its top and a ground rail at its bottom.
- The metallization technology information to perform the routing steps. Typically, in 65mn there are up to 10 metal layers with various thicknesses and widths (i.e., R and C).
- The standard cell library timing information to perform the timing analysis.

Ultimately, the PnR tool will issue the following files[14] :

- The GDSII layout
- A post PnR Verilog Netlist which is different from the gate netlist :
    - It can contains re-optimizations of the netlist by the synthesis tool embedded in the PnR tool. This can happen if the design does not meet the timing constraints.

[11] https://www.wikiwand.com/en/Value_change_dump
[12] http://www.truevue.org/p/919
[13] https://en.wikipedia.org/wiki/GDSII
[14] Not all these files are mandatory to generate, though we do generate them in this lab because we propose to use them in the flow. You may need to generate other files if you need them.

    - It contains the clock tree. The clock is not idea anymore but now physically spread in the circuit.
    - It contains buffers which have been inserted to compensate for hold violations correction (on fast paths) or setup time violation correction (on slow paths).
    - The drive of some gates may have been optimized to meet the timing constraints or to reduce the leakage.
    - Various other types of modifications.
- A timing information SDF file. This file is highly similar to the SDF file generated post synthesis, but does now cover for the timing data of the post PnR. The SDF is interpreted by the simulator which does only understand timing.
- A Parasitic data file (Standard Parasitic Exchange Format – SPEF[15]) which contains the R and C values of the metals between the logic gates. This file may seem redundant with the SDF format, and it somewhat is, though, it does not contain exactly the same information, and will not be used for the same purpose. The SPEF file can be interpreted by the power estimator which needs R and C models to extract the power consumption of the wires.

2.4.2. POST PLACE-AND-ROUTE LOGIC SIMULATION

As for post synthesis, the PnR Verilog gate-level netlist can be finally simulated using the existing VHDL testbenches and the SDF data extracted from the layout. This simulation will allow the designer to ensure that their design works correctly post PnR.

You may start to feel that verification is an important part of the microelectronics industry. As a designer you **MUST** consider that anything which has not been tested **DOES NOT WORK**. Thereby, any design step you do, will generally be tested and verified with different tools and different sets of technology inputs. The main reason being that the more complex the design is, the more abstract are the models used. More abstract models take approximations which combined along the stack, may lead to a non-working design.

In such an environment. Cross-verification is mandatory, particularly when scaling to smaller technology nodes. As an example, for universities test chips, the fabrication of a 9mm2 chip in a 12nm technology node technology (already a 5years old process) costs around 300.000USD[16] for 100 chips. In this context, you'd better verify any design step.

2.4.3. MORE VERIFICATION ! THERE IS NEVER TOO MUCH VERIFICATION

Power Analysis

In this lab, we make you use a power analyser called PrimeTime from Synopsys. It takes the different netlists (post synthesis or the post PnR), the parasitic data, the standard cell and IPs timing and power info, and the switching activity of the circuit under test, to estimate the power consumption in this case. Synthesis and PnR tools can estimate the power consumption, but this will be done assuming a 50% activity on the inputs, which is unrealistic.

[15] https://en.wikipedia.org/wiki/Standard_Parasitic_Exchange_Format
[16] https://europractice-ic.com/schedules-prices-2023/

Physical Verification : Design Rule Check (DRC)

Yes, you know this tool, Calibre. You did use it to verify your layout during the phase 1. Besides having one more layer of independent verification, you want to use this tool for the following reasons :

- The PnR tools know the routing techniques from the tech lef (we will explain that later). This file may be outdated, not correct, or may lead to situations where DRC rules are not respected. The foundry provides a set of rules which are certified for fabrication. Running the layout through a certified DRC tool with a certified set of physical rules is the only reliable way to know that the design is correct.
- The different IPs and standard cells you use in your design are abstracted in the PnR tools in the form of lef files, which represent the physical footprint of the design. The lef files have been generated by the design team who did design the IPs, or by an automatic memory compiler. **How much do you trust these ? this is a trap question, you DON'T**. You cannot ensure that the lef is correct and that it will not induce DRC errors when co-integrated with metal and standard cells ! You can also note that lef files can be purposely simplified, thereby making them not always trustworthy.
- Do you trust the integrated DRC checker of the PnR tool ? (again you don't because it does not know everything, and when using the tech lef, it can make approximations).

Physical Verification : Layout Versus Schematic (LVS)

Same story. You want to verify here that the generated circuit is equivalent to the circuit you verified in the Post PnR simulation (post PnR netlist). How do you ensure that the PnR tools did not do short circuits? Same comment about the lef file, if not correct, it could lead to short circuits. The PnR tool has some connectivity checks embedded, but it's not qualified by the foundry for sign-off. You want to use the certified rules for LVS.

Timing verification – not done in this series of LABs.

How do you ensure that the timing analysis ran by the PnR tool is correct ?

Through the simulation ? it is not enough as the SDF may be incorrect. For this, you want to use tools that will evaluate the timing of the different paths, and ensure that there are no hold and setup violations.

> *Hold violations are critical as these cannot be corrected after fabrication.*

Spice verification

This is the ultimate level of verification one could run. Transforming the circuit in a spice netlist (transistor level) and simulate it pre or post-PEX as you did in phase 1. Though, you may foresee that for fairly complex circuits, this is extremely complex and long to run. These kind of verifications can be performed at the scale of small circuits (a few 10.000 of gates), but not for complete circuits.

2.5. RTL DESIGN EXAMPLE USED IN THIS LAB

In this lab, we propose to explore the design flow through a design example which can be modified easily. The goal of the class is not to iterate on a RTL code, but rather to understand how the code can be related to the circuit being generated at the end. Thereby, we here describe the example and the different components as well as its function, though, you will generally not need to heavily edit it.

On the other hand, you will eventually navigate through different possible implementations of it, in order to explore design trade-offs, generally called PPA (Performance, Power, Area).

2.5.1. INITIAL IMPLEMENTATION

The proposed circuit is a simple arithmetic unit which contains :

- Several memories (MEMA0, MEMA1, MEMB0, MEMB1, MEMO0 and MEMO1).
- An Adder/Subtractor (ADD/SUB) taking 32bits words as input.
- A 32bits multiplier (MULT32b) taking as input 32bits words (the 32LSBs from MEM0 and MEM1 outputs).The output is a 64bit word.
- A set of control and Status registers. Which can be set by the testbench, in order to control the different operations to be performed. At every rising edge of the clk signal, all the registers are updated. When rst is high, they are all reset to some initial conditions.
- A scheduler, which is essentially a big finite state machine scheduling all the operations in the arithmetic unit based on the values of the different control registers. It also controls the multiplexers (select signal) and the memory operations (enable, read and write signals).

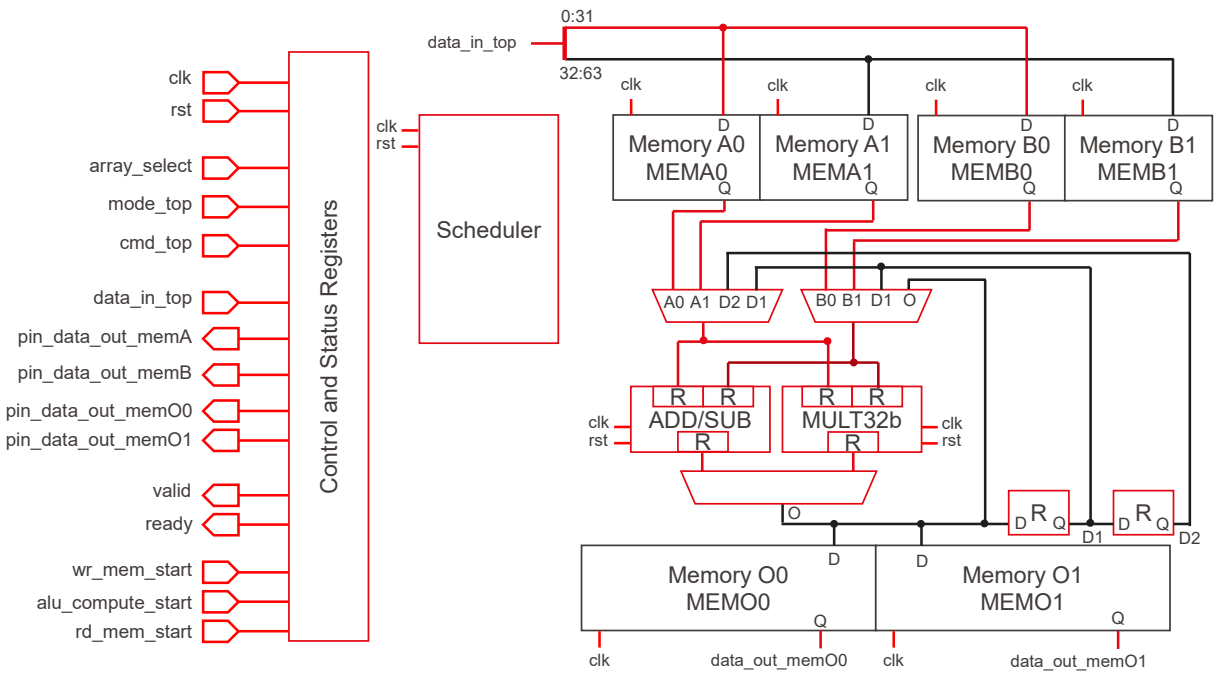

Figure 2: 32-bit RTL design example used in this series of labs.

The design performs 4 operations on a set of data. Here is an example of such operations:

*X=A0*B0*

Y=A1+B1

O0= X+Y

*O1= X*Y*

2.5.2. VARIATIONS ON THE IMPLEMENTATION

The proposed ALU can be implemented in different ways. Each implementation can perform the same workload while optimizing the PPA metric in a different way.

- A serial approach : 1 ALU
  - This is the architecture presented before. Each computation is done after the previous one.
- A parallel approach : 2 ALU
  - In this architecture, the computation is done in parallel. It contains more processing elements.
- A pipeline approach : 4 ALU
  - This architecture implements concurrency, which makes the computation faster while staying parallel.

2.6. DESIGN PROJECT STRUCTURE

Given the number of EDA tools and files used in the flow, the working environment must be organized in a proper way. To that end, a *design project directory structure*[17] can be created by running the following command in a terminal shell[18]. As for the full custom labs, you will copy a reference design environment which you'll use to work on. **You should do this operation only once at the beginning of the labs to create your environment !**

```
$HOME> cp -r /education/classes/2025-2026/EE429/EE429_SEMICUSTOM
```

The tutorial installation creates a design project directory structure rooted at the directory `EE429_SEMICUSTOM` [19]:

```
EE429_SEMICUSTOM/
├── DesignCompiler - - - Synopsys design compiler workspace for logic synthesis
├── edadk.conf     - - - configuration file used in EPFL for EDA tools
├── EMBEDIT        - - - Memory compiler folder – used to generate memory IPs
├── HDL            - - - VHDL/Verilog source files
├── IPS            - - - Hardware IPs : standard cells, memories
├── LibraryCompiler- - - Synopsys Library compiler folder – used to compile lib into db files
├── PrimeTime      - - - Synopsys Prime Time working folder
├── INNOVUS        - - - Cadence Innovus working folder
├── VIRTUOSO       - - - Cadence Virtuoso working folder
└── QUESTASIM      - - - Siemens QuestaSim logic simulator folder
```

[17] This is the global design project. Each EDA tool may define its own editing, simulation, or synthesis projects.

[18] `$HOME>` is the prompt at the shell command line; it should not be typed in. It indicates, and should be actually replaced by, the name of your home directory. The working environment can be actually installed in any suitable location in your home directory.

[19] This is actually not the only way to organize files. Each design team should define its own one and stick to it.

The necessary files for using the GlobalFoundries 65nm CMOS design kit for semi-custom design are installed in appropriate project subdirectories. The roles of the main project subdirectories are as follows.

The `HDL` directory contains all VHDL and Verilog source files created by the user or generated by the tools:

```
HDL/
├── GATE                        - - - post synthesis Verilog netlists
├── PLACED                      - - - post place and route Verilog netlists
├── RTL_SERIAL                  - - - RTL VHDL files for the serial implementation
│   ├── ALU_32b.vhd
│   ├── alu32_pkg.vhd
│   ├── mult_32b.vhd
│   ├── top_32b.vhd
│   └── top_scheduler.vhd
├── RTL_PRALLEL                - - - RTL VHDL files for the parallel implementation
│   ├── top_32b.vhd
│   └── top_scheduler.vhd
├── RTL_PIPELINE               - - - RTL VHDL files for the pipeline implementation
│   ├── top_32b.vhd
│   └── top_scheduler.vhd
└── TBENCH                      - - - testbenches used for simulation
    └── top_32b_tb.vhd
```

The `IPS` directory contains intellectual property files such as the standard cell library provided by the design kit or externally generated blocks such as RAM/ROM memories or register files. Here two folders are present. Memories and STDCELLS. One configuration of memory is available at the moment. Note that the two standard cell flavors hd ans hs are actually symbolic links to /dkits/synopsys/DesignWare_logic_libs/commonplatform65nlp/. This is because these files are stored in a secured file server and should NOT be copied.

```
IPS/
├── MEMORIES
│   └── sramHD_32x32
│   └── sramHD_32x64
│   └── sramHD_64x64
└── STDCELLS
    ├── hd -> /dkits/synopsys/DesignWare_logic_libs/commonplatform65nlp/hd
    └── hs -> /dkits/synopsys/DesignWare_logic_libs/commonplatform65nlp/hs
```

The `QUESTASIM` directory contains all files required to perform logic simulation using the SiemensEDA/Mentor QuestaSim tool:

```
QUESTASIM/
├── ACTIVITY                      - - - folder for VCD activity files
├── edadk.conf -> ../edadk.conf   - - - configuration file used in EPFL for EDA tools
├── IPS -> ../IPS/                - - - link to IPS folder
├── LIBS                          - - - folder for compiled stdcells
├── HDL -> ../HDL/                - - - link to VHDL/Verilog source files
└── modelsim.ini                  - - - configuration file for questasim
```

The `DesignCompiler` directory contains all files required to perform RTL synthesis using the Synopsys Design Vision/Compiler tool:

```
DesignCompiler/
├── BIN                          - - - folder for tcl scripts
├── DB                           - - - folder for design databases
├── DLIB                         - - - folder for design library temp file storage
├── edadk.conf -> ../edadk.conf  - - - configuration file used in EPFL for EDA tools
├── HDL -> ../HDL/               - - - link to VHDL/Verilog source files
├── IPS -> ../IPS/               - - - link to IPS folder
├── RPT                          - - - folder for hosting reports
├── SDC                          - - - folder for SDC files
└── TIM                          - - - folder for SDF timing files
```

The `PrimeTime` directory contains all files required to perform power analysis using the Synopsys PrimeTime/PrimePower tool:

```
PrimeTime/
├── BIN                           - - - folder for tcl scripts
├── edadk.conf -> ../edadk.conf   - - - configuration file used in EPFL for EDA tools
├── IPS -> ../IPS/                - - - link to IPS folder
├── SDC -> ../DesignCompiler/SDC  - - - link to SDC files from DesignCompiler
└── WAVEFORMS                     - - - folder for generated waveforms
```

The two other folders , Embedit and Library compiler will be discussed later on.

Each tool must be run in its own subdirectory, otherwise it may not work properly. Each tool subdirectory contains important files such as setup, configuration, and library files. Some files are actually symbolic links to other locations to save disk space and avoid multiple copies of secured files. Another reason to have one subdirectory per tool is to keep the many generated files whose output cannot be (easily) controlled in well-defined places so the project structure is not unnecessary cluttered.
Note also the existence of the `edadk.conf` file, which contains the information on the EDA tools and the design kit used in the project (providers, names, versions). It is centrally defined in the project root directory and linked from tool subdirectories. This file is specific to the EPFL infrastructure for EDA, each company has its own way to manage EDA tools.

- *One proper way of working in this design project structure is to create one terminal window per tool, each terminal window having its current working directory in a tool directory. This way, you can safely run each tool in its proper environment. Another way would be to use different Linux workspaces for each tool/task – we explain how to do so section 6.*