Integrated Systems Architectures

Lab Guidelines

The following guidelines have to be intended as a workflow to stick to whenever a group has some difficulties in pursuing the lab requirements. The process doesn't want to make students' lives more complicated, but rather promote self-learning in all its forms by having access to countless open source websites and books. Moreover, the student must consider the large number of his colleagues, in sharp contrast to the number of teachers, which is why it is not possible to provide advice to all groups on a continuous and exhaustive basis.

Please stick to the following points without exception and with the utmost commitment, as teachers will undertake to provide support whenever the practice is followed correctly.

TO DO LIST

1. **Use your brain**: before you open your laptop make sure that the whole project is clear in your mind and in your group's. Designing means exploring the space of the solutions to find the one that best suits your target;
2. **Read carefully the lab requirements**: If you don't understand some of them, read on, the answer is probably later. it is the teacher's duty to provide details of laboratory experience;
3. **Exploit manuals and books**: you will find yourself using complex commercial tools widely used in companies around the world. These tools are accompanied by manuals easily found online along with countless forums. If you want to deepen your knowledge or solve some problem this is a good starting point;
4. **Google your errors and doubts**: refer to the previous point;
5. **Confront with your peers**: Sometimes when you have a problem the best solution is to talk to your colleagues. Reasoning together can help you find alternative solutions;
6. **Attend lab and tutoring slots**: The course schedule has 3 hours of lab on Tuesday and 1.5 hour of counseling on Thursday. These are the slots to solve doubts.
7. **Mail**: If all the previous points fail to help you finding a solution to your problem, you are free to send emails to solve doubts out of the official slots, the teachers are free to answer or not.

The above list has to be followed in order.

TO BE AVOIDED

- Ask questions about information clearly present in the lab requirement text;
- Report with a non-working testbench without knowing where the problem is and ask for debugging help. Laziness is not tolerated, if the architecture does not work it is up to you to find the problem, or at least identify which component is responsible before asking for help;
- Asking for counseling skipping the previous points.

N.B. All lab experiences are **NOT** mandatory, it is up to the group to decide how many of them to carry out according to the rules provided at the bottom of this document.

References:
Kristjane Koleci, VLSI lab, DET, lab phone: 4004
Mail address:
kristjane.koleci@polito.it

Emanuele Valpreda, VLSI lab, DET, lab phone: 4004
Mail address:
emanuele.valpreda@polito.it

RULES and DEADLINES

Lab activities:

1. First lab, filter architecture and implementation.
2. Second lab, digital arithmetic.
3. Third lab, RISC-V.
4. Fourth lab, verification.

Labs activities are OPTIONAL and can be accomplished in **three modes**:

1. Standard:
   (a) 3 labs;
   (b) up to 3 points each respecting deadlines. Up to 1.5 points each after deadlines. Lab discussion must be within the winter exam session.
2. extended:
   (a) 4 labs;
   (b) up to 3 points for lab 1, 2 and 3, up to 2 points for lab 4 respecting deadlines. Up to 1.5 points for lab 1, 2 and 3, up to 1 point for lab 4 after deadlines.
   (c) In this mode lab 1, 2 and 3 are mandatory. Dropping one among lab 1, 2 and 3 makes lab 4 not evaluated.
3. Special project:
   (a) 2 labs (first and second mandatory);
   (b) up to 3 points each respecting deadlines. Up to 1.5 points each after deadlines;
   (c) Complex project substituting the third lab (up to 6 points, mandatory deadline)

Deadlines:

1. First lab delivery deadline: Nov. 21 at 23:59;
2. Second lab delivery deadline: Dec. 19 at 23:59 ;
3. Third lab delivery deadline: approximately the last written exam in the winter session;
4. Fourth lab delivery deadline: Apr. 17 at 23:59;
5. Complex project deadline: Jun. 30 at 23:59.

NOTE:the complex project might be a starting point to start getting some background for the thesis work.

Integrated Systems Architectures

Setup and login

To attend these labs you have to use a PC where an X2go client is installed or you have to install an X2go client on your PC. Once the PC is ready you have to configure X2go.

1 Connection

From the X2go Client window choose *Session− > New session.* Input the following values in the *Session* tab:

Session name: led-isa

Host: led-x3850-2.polito.it

Login: < *your user* >

Port: 10038

Session type: MATE

If you want in the tab *Shared folders* you can specify the path of a local folder of the host PC, which will be mounted on the server in your home directory under *media.* Please note that with this configuration you can access the ISA server from the polito network as well as from home.

2 Known issues

1. Only one session per group is allowed lo limit the server load;
2. You can suspend your session and at the next connection you will find the same session you left;
3. To transfer files from/to this system you can use one of the following possibilities:
   - your email at Politecnico di Torino;
   - portale della didattica via the Virtual Disk (Disco Virtuale) in your student account;
   - the X2go *Shared folders* option.

Integrated Systems Architectures

General suggestions

1 Test Bench

This document is to be intended as a quick tutorial on HDL test benches. Nowadays, test benches are heavily used in companies to validate every single component. Entire company sections are dedicated to test and validate what comes from the design section.
During this course you will face two different situations: given specifications, you design a component and a test bench is provided to test your architecture; given specifications you must design both the architecture and the test bench.
Teachers will use automatic test bench systems to verify that the designs are correct. Considering this it becomes very important to respect the specifications and interfaces. Architectures with wrong interfaces will be considered as wrong designs.
Whenever you design a component, it is of fundamental importance to test it to ensure that its functioning is compliant to the specifications. In order to verify that a component behaves according to the project requirements, it must be subjected to a test:

- controlled stimuli are applied on the input;
- output outcomes are verified to be compliant.

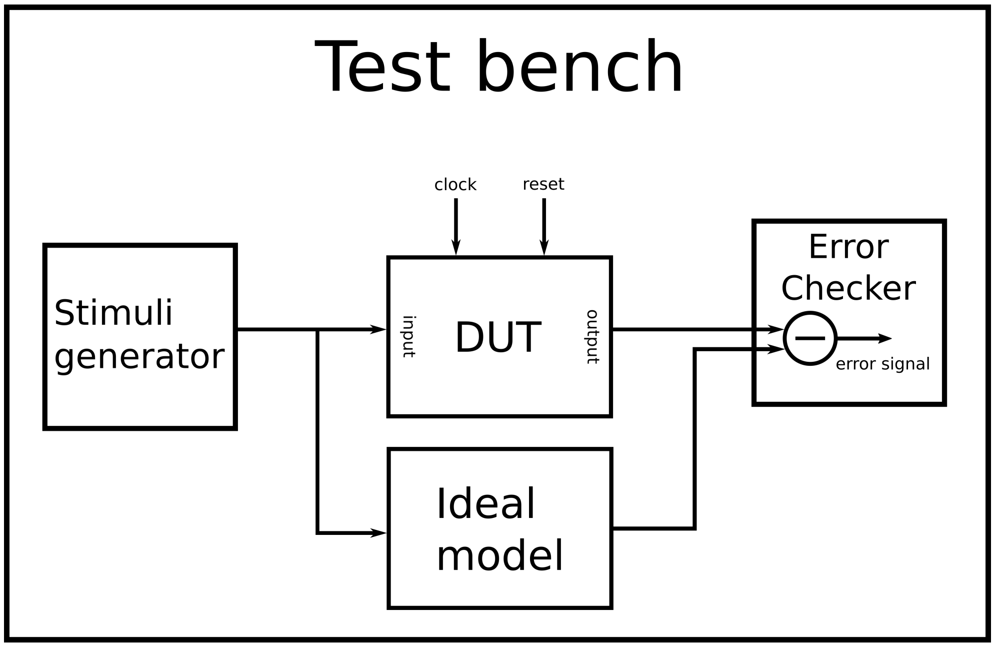

Figura 1: Schematic block of a test bench.

This VHDL operation is possible with an HDL simulator, in our case Modelsim or Questasim. An appropriate entity shall be provided to describe the test to be performed: the **unit of test bench**.
The unit of test bench shall:

- incorporate the Unit Under Test (UUT), or Device Under Test (DUT);
- contain a description of the stimuli to be applied;

HDL test bench characteristics:

- No input or output ports;
- Input signals can be varied in time, so as to cover multiple configurations;
- If combinations of input signal values are too numerous, the test is limited to more meaningful configurations;
- Signals can be described using concurrent instructions (assignment with **after** keyword) or sequential (**wait** and **wait for** in **process** instructions).

Test benches are used for test purpose only, **not for synthesis**, therefore is possible to use several VHDL constructs such as **assert**, **report**, **for loops** etc. Remember to remove all test bench files when you perform the synthesis of your circuit, otherwise, Design Compiler will report many errors, and more important to synthesize the test bench has no meaning.
In the following, a simple example of a test bench for a combinatorial component will be presented.

```
-- half_adder.vhd

library ieee;
use ieee.std_logic_1164.all;

entity half_adder is
port (a, b : in std_logic;
sum, carry : out std_logic
);
end half_adder;

architecture arch of half_adder is
begin
sum <= a xor b;
carry <= a and b;
end arch;
```

As you can easily understand the DUT is a half adder. Now let's analyze a possible implementation of the test bench usually abbreviated as "tb".

```
-- example of a simple test bench

library ieee;
use ieee.std_logic_1164.all;


entity half_adder_tb is
end half_adder_tb;

architecture tb of half_adder_tb is
signal a, b : std_logic;  -- inputs
signal sum, carry : std_logic;  -- outputs
begin
-- connecting test bench signals with half_adder.vhd
UUT : entity work.half_adder port map (a => a, b => b, sum => sum, carry => carry);

-- inputs change in time (not synthesizable)
-- 00 at 0 ns
-- 01 at 20 ns, as b is 0 at 20 ns and a is changed to 1 at 20 ns
-- 10 at 40 ns
-- 11 at 60 ns
```

```
a <= '0', '1' after 20 ns, '0' after 40 ns, '1' after 60 ns;
b <= '0', '1' after 40 ns;
end tb ;
```

As you can see the entity of the *tb* has no inputs or outputs. The reason is that this component does not communicate with other external elements. The *DUT* is then connected to local signals that vary in time, going through all the possible combinations. Analyzing how the outputs of the DUT vary one can understand if it is performing correctly.

What happens when we have a very complex system? Do we have to check the behavior of every single signal?

No, we do not. When we test very complex device one trick is to generate an **error signal** that is triggered when something goes wrong.

How?

When you design a hardware device, it typically derives from a mathematical model or an algorithm. To test means to check that the model and the behavior of the HDL description are equal. So we have results coming from the HDL simulation and the ideal model. The error signal can be easily generated by a simple difference between such results.
You can exploit three different approaches:

- Implement the ideal model directly in the test bench by describing it in a behavioral manner. The error signal would be part of the tb as a difference of the data produced by the DUT and the model;
- Implement the ideal model exploiting another language such as Python, Matlab, etc. and save the results in a text file (or similar). In the tb is possible to read the text file and generate an error signal as the difference of the data produced by the DUT and the model;
- Save the data produced by the DUT on a txt file (or similar) and exploit external software to check for errors.

N.B. The previous example implement a tb for a combinatorial circuit. For sequential circuits you need also to create a **reset** and a **clock** signals. Here an example:

```
-- Process for generating the clock
Clk <= not Clk after ClockPeriod / 2;
```

Clk is a signal that every ClockPeriod / 2 changes its state.

To deepen you knowledge you can visit:
`https://vhdlguide.readthedocs.io/en/latest/vhdl/testbench.html`
`https://www.unirc.it/documentazione/materiale_didattico/599_2008_94_2798.pdf`

2 Scripting

Almost all the EDA tools used in this course supporte scripting. The use of the GUI is helpful as a first step to understand the flow, but then to repeat the flow (or part of the flow) scripts help in speeding up. The main language used in EDA tools scripting is TCL. However, for our purpose it becomes almost a sequence of commands. You can find the commands with all the details about paramenters in the documentation of each tool. However, it is possible to derive the commands by checking the console or the log file of the command-line shell, which is available in the GUI of the tool.

Example:

```
analyze -f vhdl -lib WORK ../src/file1.vhd
analyze -f vhdl -lib WORK ../src/file2.vhd
analyze -f vhdl -lib WORK ../src/top.vhd
elaborate top -lib WORK
create_clock -name My_CLK -period 5.0 CLK
compile
report_area >> ./report_area.txt
report_timing >> ./report_timing.txt
```

3 Homework report preparation

On "Portale della didattica" you have a template for the homework report. Please follow the scheme already provided and remeber that the report should be self-contained, complete, clear and accurate. Diagrams, graphs and figures which could be useful to better understand your design have to be included in the report.

Note: Please be ready to present your work during the oral discussion. You have to explain your work, the results you obtained and compare the different solutions.

4 Projects delivery

Each project you are preparing for this course must be developed and deliverd with *git*. Please refer to the *versioning* section for details about versioning and git.

5 Disk space

As the number of groups is large and the amount of disk space is finite, please do not leave large files/folders on the server. When the disk is full nobody can connect to the server.

CHAPTER 1

Versioning

In computer science, version control (versioning) is the management of multiple versions of a set of information. It is mainly used to develop engineering or computer science projects to manage the continuous evolution of digital documents such as software source code, technical drawings, text documentation, and other relevant information on which a team works. Version control systems are a software tool category considered necessary for software development teams to manage code changes over time. The version control software relies on special type databases (incremental-based) to track any code changes. If any team member makes a mistake, the developers can go back to the previous version and, by comparing it with the current one, find the problem minimizing the time and teamwork.
The source code represents a repository of valuable knowledge about a specific problem domain perfected with great effort for the team. The versioning protects the source code from catastrophic events and team members' errors. The source code is organized in a folder structure with a tree-like topology. Team members can work at the same time on different parts of it. In this scenario, version control is fundamental to track code changes and avoid any type of conflict. In fact, if more than one member modifies a file, a conflict occurs. The versioning system raises an error, but developers have to fix the issue. In this case there are two possibilities:

- variations are in the same file, but in different portions of the code, somehow decoupled from each other. In this case the developers can merge the changes.
- variations are in the same file and in same code portions. In this case it is necessary to understand if changes can coexist, or code must be rethought.

Note: every time a commit operation is performed, it is **mandatory** to insert a meaningful comment that motivates the changes and helps to trace them.

Most Version Control Systems (VCS) involve a certain vocabulary that must be known:

- **Repository (repo)**: The database storing the files;
- **Server**: The computer storing the repo;
- **Client**: The computer connecting to the repo;
- **Master**: The primary location for code in the repo. The master is the main line;
- **Branch**: Banch is a different line from the Master. The user can work in parallel on the Master and the Branch and then eventually merge them. When a Branch is created, it represents a copy of the Master;

- **Merge**: Apply the changes from one file to another, to bring it up-to-date. For example, users can merge features from one branch into another;
- **Add**: Put a file into the repository for the first time, i.e. begin tracking it with Version Control;
- **Head**: The latest revision in the repo;
- **Conflict**: When pending changes to a file contradict each other (both changes cannot be applied).

There exist mainly two types of VCS: centralized and distributed. In the former, the main repository is located on a remote server, and the user can download just a copy of the data locally, called working copy. As soon as one of the co-workers commits a change, the others can update their working copies and see the modifications. In the latter one, instead, users download the whole repository on a local workstation. This does not contain just the data, metadata, and history that allow trace file changes. When a user commits his changes, these are transferred only in the local repo, co-workers cannot see them. Only after he pushes the committed changes on the remote server co-workers to have access to them. In case of a catastrophic event on the server, the distributed VCS can recover the whole repository from a generic user. Two examples for such VCS types are Subversion for the centralized and GIT (or GitHub) for the distributed one.

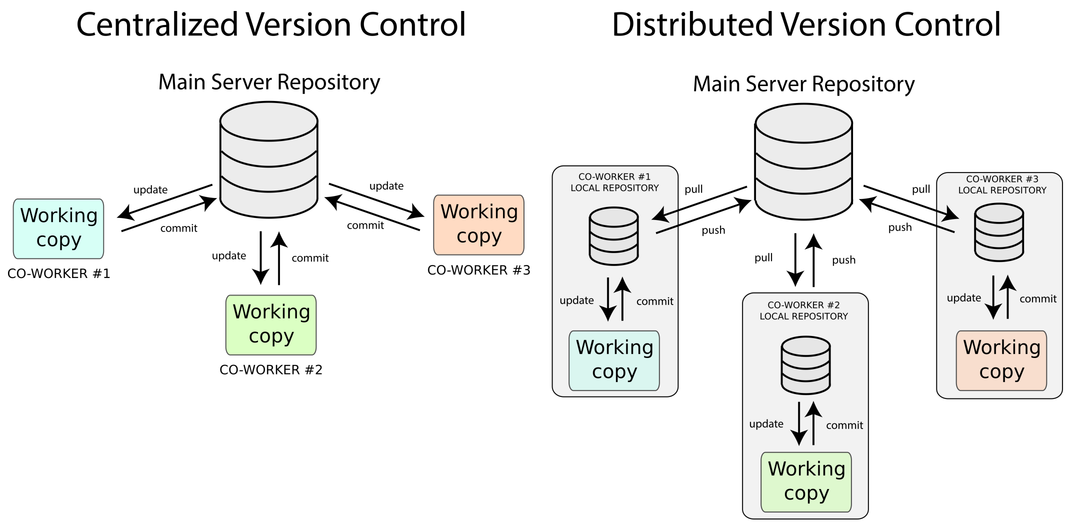

Figure 1.1: Centralized vs Distributed.

1.0.1 Versioning Benefits

It is possible to sum up the main benefits of a version control system in 3 points:

- A **long-term variation history** of each file. Such history includes author, date, and purpose of every created, deleted and altered file present in the repository. Having a so detailed chronology enables the user to turn back the clock anytime a bug occurs, identifying the root of the issue with a clear procedure, even in case of very complex projects. To be noted that different versioning software manage file differently.
- Team members can **work concurrently** on different code streams. In fact two or more branches can be developed in parallel and successively merged. This allows to have several workflows that can eventually converge. Whenever a merge is done, the software is in charge of detecting conflicts by relieving the user of this task.

- **Code traceability**. Being able to track any code changes along with their annotations and comments provides users with a powerful mean able to locate bugs. In long term program, such annotated history helps developers, even the new ones involved in the project, to easily understand the code and to take the most suitable move according to the intended design plan.

1.0.2 GIT

GIT is a distributed control version used by many big companies. In GIT many operations are local, in the sense that they need just local files to be performed. In fact since it is distributed, each user has its repository and the related history. Such an important feature allows users to work even when they are offline or they have no access to the server. GIT is based on four main stages in which files can reside: **untracked**, **modified**, **staged** and **committed**.

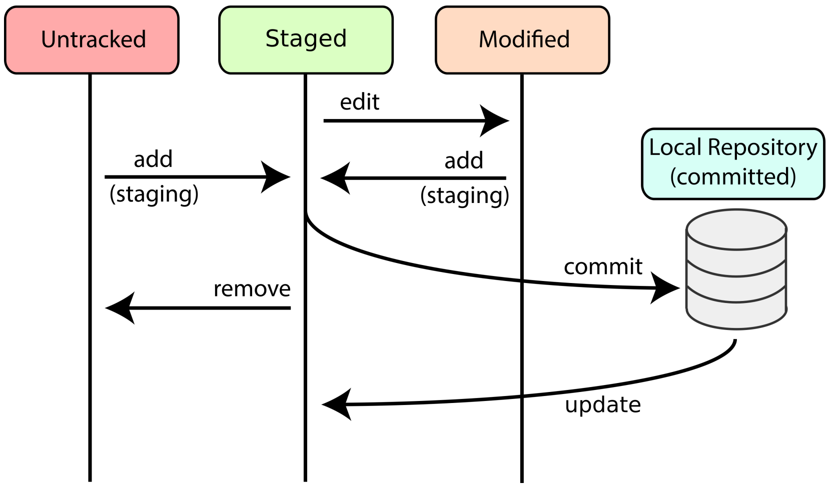

Figure 1.2: GIT stages.

Files labeled as untracked are part of the working directory, but the versioning system does not care about them, so their history is not tracked. When they are added to the repo, they are labeled as staged and the VCS starts to track their movements. Whenever a file is modified, it is labeled as modified and in order to be committed on the local repo, must be staged before. Files that have been committed can be successively pushed on the remote repository.
The working directory is a checkout of one version of the project. Such directory contains pulled files from the GIT compressed directory. Here the user modifies, deletes or creates files. The staging area is represented by a file, contained in the GIT directory that stores info about those files that will go into the next commit. GIT directory stores GIT metadata and represents the project database. Whenever a user calls for a clone, here is where files are moved.
In the following some basic commands are listed in order to get going with GIT.

<u>Local repository</u>

- **git init [project-name]** creates a new local repository in a sandbox folder;
- **git add [file]** adds (stages) [file] in preparation for versioning;
- **git add -u** adds all files already present in the repository in preparation for versioning;
- **git reset HEAD [file]** to unstage a file;
- **git checkout –[file]** discard changes in [file] and restore it as the last committed version;

- **git commit** records all staged files permanently in version history;
- **git commit -m "[message]"**;
- **git commit –amend** correct the last commit;
- **git status** describes the current situation of the repository;
- **git log** shows history of the current repository;
- **git diff** shows the differences between the files in the repository and last commit;
- **git diff [commit1] [commit2]** shows differences between [commit1] and [commit2];
- **git rm –cached [file]** removes file from version control (does not delete the file);
- **git rm [file]** deletes the file (almost definitively);
- **git reset [commit]** undoes all commits after [commit], local files are left unchanged;
- **git reset –hard [commit]** restores everything to the specified [commit].

Remote repository:

- **git clone [remote]** gets all data from the remote repository and assigns to it the shortname origin;
- **git init –bare** in a remote folder, creates a remote repository;
- **git remote add [shortname] [user@address:path to folder]** from the sandbox directory, links to the remote repository [user@address:path to folder] and assigns optional [shortname];
- **git pull [remote] [branch]** downloads most recent missing data from the remote to the local repository and automatically merges changes. [remote] can be a whole address or a shortname;
- **git fetch [remote]** downloads most recent missing data from the remote to the local repository. [remote] can be a whole address or a shortname;
- **git merge** manually merges files;
- **git push [remote] [branch]** example: git push [origin] [master] puts your changes in the remote repository. You must be up to date.

Branches:

- **git branch** lists all existing branches;
- **git branch [branch name]** begins new branch;
- **git checkout [branch name]** moves HEAD to point to the [branch name] branch;
- **git checkout -b [branch name]** creates branch [branch name] and moves HEAD to it;
- **git merge [branch name]** if you are on another branch (e.g. master) it merges [branch name] and the master branches;
- **git branch -d [branch name]** deletes branch [branch name];
- **git fetch [remote]** fetches everything from [remote] and sets remote-tracking branch [remote]/[RemoteBranch];

- **git checkout -b [NewBranch] [remote]/[RemoteBranch]** creates [NewBranch] from [RemoteBranch] and moves head to it.

For further details please visit the website `https://rogerdudler.github.io/git-guide/index.html`

Integrated System Architectures

FREQUENTLY ASKED QUESTIONS

Maurizio Martina

PLEASE: read carefully the pdfs available on "Portale della Didattica", several answers to your problems are already written in the documentation. Moreover, the tools very often give you important information on the command line with some details about errors, warnings, … please read them.

Q) The tool (simulator, synthesizer, …) does not start.

A) Under /software/scripts you find <u>all</u> the initialization scripts, issue the command

```
source <init_file>
```

each of these script files sets-up the environment for launching the tool

Q) Design Compiler does not find the target library/components.

A) Make sure you have created the ".synopsys_dc.setup" file (the name starts with the '.', dot, character). If you copy&paste the text from the pdf be aware of underscore '_' and space characters.

Q) The tool states it cannot create the file (netlist, saif, vcd, …)

A) Usually this happens when you try to write a file in a directory and the directory does not exist. Please check/create the directory before issuing the command.

Q) Encounter is not able to find the VDD and GND pins.

A) Usually this happens when you create from the scratch your configuration file. You have to download it from "Portale della Didattica" and modify it instead.

Q) The tool cannot find a file or an element I created, but the element exists.

A) Usually, this happens if you mix the case of characters when naming files, data, components, elements, … Please note that VHDL is not case sensitive, whereas Verilog is. As a good design rule avoid mixing the case of characters, choose your own rules and be consistent with them.

Q) Modelsim is not accepting all the options for switching activity annotation.

A) The options passed to *vsim* must be all on the same command line. In the pdf they appear on different lines just due to the limited number of characters per line the document style can handle.

Q) Modelsim is not able to compile/simulate the SystemC files.

A) Please check you sourced the correct 'init_file' for modelsim. Not all modelsim versions support SystemC, only the most recent ones.

Q) The command(s) shown in the pdf give an error

A) Probably you made a 'Copy-Paste' from the pdf to the shell/tool/editor, please check the characters are correct (especially underscore '_' and spaces).