Introduction to FPGA programming with HLS

Mirko Mariotti

June 2026

Outline

Course Overview

Course structure:

- Lectures: theoretical concepts and practical examples
- Labs: hands-on exercises using Xilinx Vitis HLS and Pynq

Pre-requisites:

- Familiarity with **Python** and **C/C++** programming

Other pre-requisites (need a refresher?):

- Basic understanding of **Linux command line** and development environment (e.g., terminal, text editors, compilers)
- We also will be using **Conda** for environment management, **Makefiles** for build automation, and **Git** for version control.

Hands-on structure

- Each Hands-on project will have a specific name and a clear set of objectives.
- We will start with simple designs and gradually increase complexity as we progress through the course
- Each project will have its own directory within the course repository, containing all necessary files and instructions for completion.
- Each project will include a checklist of tasks to be completed

Github Repository

git clone https://github.com/mmirko/FPGALab.git

Introduction to Linux CLI

- The **Command Line Interface (CLI)** is a text-based interface for interacting with the operating system
- Also called: terminal, shell, console, or command prompt
- Most common shells:
  - `bash` (Bourne Again Shell) - most popular
  - `zsh` (Z Shell) - feature-rich alternative
  - `sh` - original Unix shell
- Why use the CLI?
  - More powerful and flexible than GUI
  - Essential for remote server management
  - Automation through scripts
  - Faster for many tasks once learned

File System Navigation

Basic Navigation Commands:

- `pwd` - print working directory
- `ls` - list directory contents
- `cd` - change directory
- `mkdir` - make directory
- `rmdir` - remove empty directory

Examples:

```
$ pwd
/home/user

$ ls -la
total 48
drwxr-xr-x  5 user

$ cd Documents

$ mkdir project

$ cd ..
```

Special paths: `.` (current dir), `..` (parent dir), `~` (home dir), `/` (root), `-` (previous dir)

File Operations

Creating and Viewing:

- `touch file` - create empty file
- `cat file` - display file content
- `less file` - view file page by page
- `head file` - show first lines
- `tail file` - show last lines

Copying, Moving, Deleting:

- `cp src dst` - copy file
- `cp -r dir1 dir2` - copy directory
- `mv src dst` - move/rename
- `rm file` - remove file
- `rm -r dir` - remove directory

Example

```
$ touch newfile.txt
$ cp newfile.txt backup.txt
$ mv backup.txt /tmp/
$ rm newfile.txt
```

File Permissions

Linux uses a permission system to control access to files and directories.

Permission Format: rwxrwxrwx

```
-rw-r--r--  1 user group  1024 Jun 07 10:30 file.txt
 |||-------|-------others (read, write, execute)
 ||--------group (read, write, execute)
 |---------owner (read, write, execute)
```

Commands:

- `chmod 755 file` - change permissions (rwxr-xr-x)
- `chmod +x script.sh` - add execute permission
- `chown user:group file` - change owner/group

Text Processing and Searching

Search and Filter:

- `grep pattern file` - search text
- `find path -name "*.txt"` - find files
- `wc file` - word/line count
- `sort file` - sort lines
- `uniq file` - remove duplicates

Examples:

```
$ grep "error" log.txt

$ find . -name "*.c"

$ grep -r "TODO" .

$ cat file | sort | uniq

$ wc -l *.txt
```

Common grep options: `-i` (ignore case), `-r` (recursive), `-n` (line numbers), `-v` (invert)

Environment Variables and Configuration

Environment variables store configuration information accessible to programs.

Common Variables:

- `PATH` - executable search paths
- `HOME` - user's home directory
- `USER` - current username
- `SHELL` - current shell
- `PWD` - working directory

Commands:

```
$ echo $PATH
$ export VAR=value
$ env
$ printenv

# Add to ~/.bashrc
export PATH=$PATH:/new/path
```

What is a Makefile?

- A **Makefile** is a special file that automates the build process of software projects
- It contains a set of directives (rules) that define how to compile and link programs
- The `make` utility reads the Makefile and executes the appropriate commands
- Benefits:
  - Automates repetitive compilation tasks
  - Rebuilds only what has changed (incremental builds)
  - Manages dependencies between files
  - Platform-independent build automation

Basic Makefile Structure

A Makefile consists of rules with the following syntax:

Rule Syntax

```
target: dependencies
 command
 command
```

- **target**: the file to be created or action to perform
- **dependencies**: files that the target depends on
- **command**: shell commands to execute (must be indented with TAB)

Important: Commands must be indented with a TAB character, not spaces!

Simple Makefile Example

Example: Compiling a C Program

```
CC = gcc
CFLAGS = -Wall -O2

program: main.o utils.o
 $(CC) $(CFLAGS) -o program main.o utils.o

main.o: main.c
 $(CC) $(CFLAGS) -c main.c

utils.o: utils.c utils.h
 $(CC) $(CFLAGS) -c utils.c

clean:
 rm -f *.o program
```

Makefile Variables and Patterns

Variables:

- Define with `VAR = value`
- Use with `$(VAR)`
- Common variables:
  - `CC`: compiler
  - `CFLAGS`: compiler flags
  - `LDFLAGS`: linker flags

Automatic Variables:

- `$@`: target name
- `$<`: first dependency
- `$^`: all dependencies
- `$*`: stem of pattern

Pattern Rule Example

```
%.o: %.c
 $(CC) $(CFLAGS) -c $< -o $@
```

Common Makefile Targets and Best Practices

Standard Targets:

- `all`: build everything (default)
- `clean`: remove generated files
- `install`: install the program
- `test`: run tests
- `help`: show available targets

Best Practices:

- Use `.PHONY` for non-file targets
- Add comments for clarity
- Use variables for flexibility
- Keep rules simple and readable
- Include dependency tracking

Example

```
.PHONY: all clean install

all: program

clean:
 rm -f *.o program
```

What is Conda?

- **Conda** is an open-source package and environment management system
- Works for Python, R, and other languages
- Key features:
  - Installs packages and their dependencies
  - Creates isolated environments for different projects
  - Works across platforms (Windows, macOS, Linux)
  - Manages binary packages (not just source code)
- Two main distributions:
  - **Anaconda**: full distribution with 250+ pre-installed packages
  - **Miniconda**: minimal installer with just conda and Python
- Why use Conda?
  - Avoid dependency conflicts between projects
  - Reproduce exact environments on different machines
  - Easy version control for packages

Creating and Managing Environments

Basic Environment Commands:

- Create new environment
- Activate environment
- Deactivate environment
- List all environments
- Remove environment

Examples:

```
# Create environment
$ conda create -n myenv

# With specific Python
$ conda create -n py39 python=3.9

# Activate
$ conda activate myenv

# Deactivate
$ conda deactivate

# List environments
$ conda env list

# Remove
$ conda env remove -n myenv
```

Installing and Managing Packages

Package Operations:

- Install packages
- Update packages
- Remove packages
- List installed packages
- Search for packages
- Export environment

Export/Import Environment

```
$ conda env export > environment.yml
$ conda env create -f environment.yml
```

Examples:

```
# Install packages
$ conda install numpy pandas

# Specific version
$ conda install numpy=1.21.0

# Update package
$ conda update numpy

# Remove package
$ conda remove numpy

# List packages
$ conda list

# Search
$ conda search scipy
```

What is Git ?

- Git is a distributed version control system.
- It is used to track changes in source code during software development.
- It allows multiple developers to work on a project simultaneously without conflicts.
- Git is designed to handle everything from small to very large projects with speed and efficiency.

Git Basic Concepts

- **Repository (repo)**: A directory that contains all the files and history of a project.
- **Init**: The command to create a new Git repository.
- **Clone**: A copy of a repository that is stored on your local machine.

Initialize a repository

```
cd some-empty-directory
git init
```

Clone a repository

```
git clone https://github.com/mmirko/FPGALab.git
```

Git Basic Concepts

Working inside a repository

- **Add**: Stages changes in the working directory for the next commit.
- **Commit**: Saves the staged changes to the repository with a message describing the changes.
- **Status**: Displays the state of the working directory and staging area.

Stage changes

```
git add file1 file2 ...
```

Commit changes

```
git commit -m "Commit message"
```

Check status

```
git status
```

Basic Concepts

Working with branches

- **Branch**: A separate line of development in a repository.
- **Checkout**: Switches to a different branch or restores working tree files.
- **Merge**: Combines changes from one branch into another.

Create a new branch

```
git branch new-branch
```

Switch to a branch

```
git checkout new-branch
```

Merge branches

```
git merge branch-to-merge
```

Basic Concepts

Remote repositories

- **Remote**: A version of the repository that is hosted on a server.
- **Push**: Uploads local changes to a remote repository.
- **Pull**: Fetches changes from a remote repository and merges them into the local repository.

Add a remote repository

git remote add origin remote-repo-url (automatically created when cloning)

Push changes to remote

git push origin branch-name (or git push)

Pull changes from remote

git pull origin branch-name (or git pull)

github

Hosting Git repositories

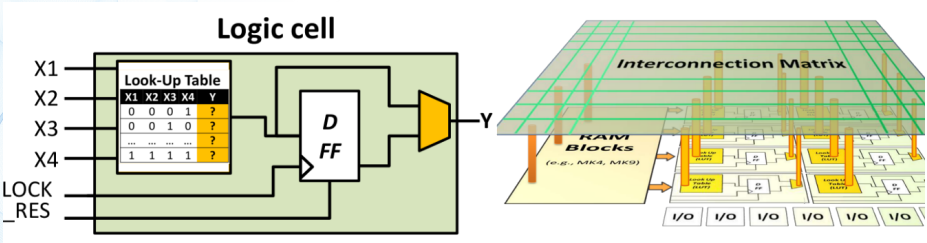
- GitHub is a web-based platform that provides hosting for Git repositories.
- It offers features like issue tracking, pull requests, and collaboration tools.
- GitHub allows developers to share their code, collaborate on projects, and contribute to open-source software.
- It provides a user-friendly interface for managing repositories and viewing commit history.

GitHub Repository for this course

`https://github.com/mmirko/FPGALab.git`

What is an FPGA?

An FPGA (Field-Programmable Gate Array) is an integrated circuit that can be configured by the user after manufacturing. It consists of an array of programmable logic blocks and a hierarchy of reconfigurable interconnects that allow the blocks to be wired together, similar to a one-chip programmable breadboard.



FPGA Programming

HDL

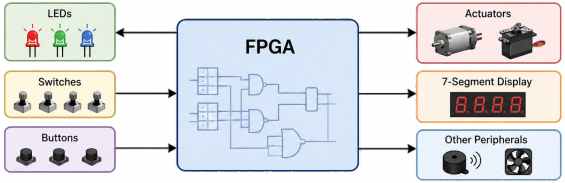
FPGA programming traditionally involves using Hardware Description Languages (HDLs) such as Verilog or VHDL. This is the topic of the companion course "Introduction to FPGA programming with HDL".

Programming an FPGA with HDL involves describing the desired hardware behavior at a low level, which can be time-consuming and requires a deep understanding of digital design principles. However, it allows for maximum control over the hardware implementation and can lead to highly optimized designs.

Once the FPGA is programmed, it can perform a wide range of tasks, and can be used in various scenarios, such as:

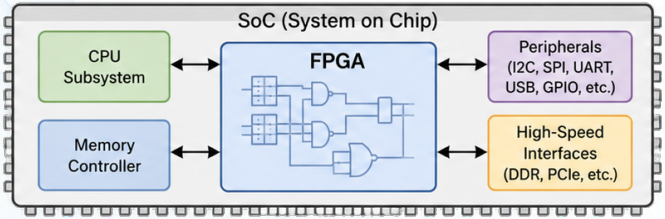
Standalone

- FPGAs can be used as standalone devices, where they are programmed to perform specific tasks without the need for an external processor.
- In this configuration, the FPGA can directly interface with sensors, actuators, and other peripherals, making it ideal for applications that require real-time processing and low latency.



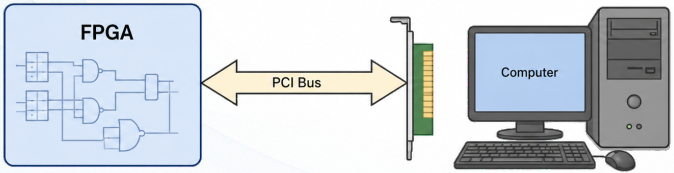
SoC

- FPGAs can also be used as part of a System on Chip (SoC) design, where they are integrated with a processor and other components on a single chip.
- In this configuration, the FPGA can be used to offload computationally intensive tasks from the processor, allowing for improved performance and efficiency.



Accelerator

- In later years, FPGAs have also been used as accelerators in data centers and high-performance computing environments.
- In this configuration, the FPGA can be used to accelerate specific workloads, similar to how GPUs are used for graphics processing, but with the added flexibility of being able to be reprogrammed for different tasks.



Accelerator

This way, FPGA enters the computing realm alongside CPUs and GPUs, providing a unique combination of flexibility and performance that can be tailored to specific applications.

FPGAs can be used to accelerate a wide range of applications, from machine learning and data analytics to scientific computing and cryptography.

Lab Setup

- We will be using the Xilinx Vitis HLS tool for high-level synthesis.
- The development environment will be set up on a Linux machine, and we will use Conda for managing dependencies and virtual environments.
- We will also use Makefiles to automate the build process and Git for version control of our projects.
- All the labs will be performed on the Alveo U55c FPGA accelerator card, which is a AMD/Xilinx product designed for high-performance computing tasks.

Lab access

You can access the lab environment remotely via Web using the provided URL and credentials.

Once logged in, you will have access to a jupyterlab environment where you can work on the labs and projects. In particular, you can use:

- The terminal to run commands and scripts
- Jupyter notebooks for interactive coding and documentation
- A complete Desktop environment for more traditional development workflows

From the lab environment, you can also access the FPGA hardware and run your synthesized designs on the Alveo U55c card.

Environment Setup

All the hands-on projects are contained in a Git repository that you can clone to your local environment

Git repository

git clone https://github.com/mmirko/FPGALab.git

The setup directory contains scripts to set up the necessary environments for Vitis HLS, Conda, and platform access.

Vitis environment

cd FPGALab/setup
source vitisetup.sh

Conda environment

cd FPGALab/setup
source condasetup.sh

Needed only once, then

conda activate fpgatest

Platform access

cd FPGALab/setup
source platformsetup.sh

Let's explore the lab environment

Core concepts

Throughput

Throughput is the rate at which a system processes data or completes operations.

- Measured in operations per unit time (e.g., samples/second, transactions/clock cycle)
- Key metric for data-intensive applications (e.g., video processing, neural networks)
- Can be improved through pipelining and parallelism

Example

A design with one new input every clock cycle. At 100 MHz, throughput = 100 million operations/second.

Core concepts

Latency

Latency is the time delay between the start of an operation and the availability of its result.

- In FPGA/HLS context: time from when input data enters a processing block until the corresponding output is produced
- Measured in clock cycles or absolute time (nanoseconds, microseconds)
- Critical for real-time applications (e.g., sensor processing, control systems)
- Trade-off with throughput: pipelining increases throughput but may increase latency

Example

A signal processing filter with 10-cycle latency means it takes 10 clock cycles from receiving an input sample to producing the filtered output.

Core concepts

Occupancy

Occupancy (or Resource Utilization) measures how much of the available FPGA resources are used by a design.

Key Resources:

- Logic Elements (LUTs, FFs)
- Block RAM (BRAM)
- DSP blocks
- I/O pins

Considerations:

- High occupancy → less room for expansion
- Low occupancy → potential for optimization
- Balanced use across resource types is ideal
- Affects timing closure and power consumption

Typical Target

Aim for 60-80% occupancy to leave headroom for routing and timing optimization.

Time vs Spatial computation

- **Time computation** (sequential execution): operations are performed one after another, reusing the same hardware resources over time.
- **Spatial computation** (concurrent execution): multiple operations are performed simultaneously using separate hardware resources.

FPGA Strength

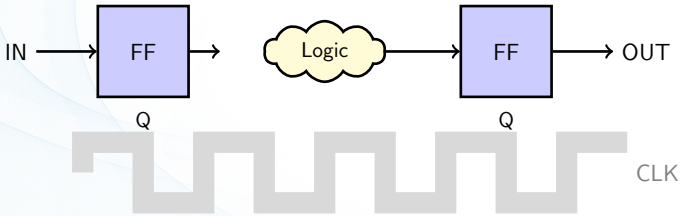
FPGAs excel at spatial computation, allowing for massive parallelism and custom data paths that can significantly improve performance for certain applications.

FPGA Resources

- **Logic Elements:** Basic building blocks (LUTs, FFs) for implementing combinational and sequential logic.
- **Block RAM (BRAM):** On-chip memory for storing data and instructions.
- **DSP Blocks:** Specialized units for efficient arithmetic operations (e.g., multiplication, addition).
- **I/O Pins:** Interfaces for connecting the FPGA to external devices and peripherals.

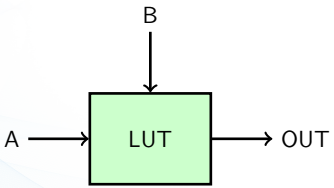
Flip-Flop

- A flip-flop is a basic digital memory element that can store one bit of information.
- It has two stable states (0 and 1) and can be used to build more complex memory structures.



LUT

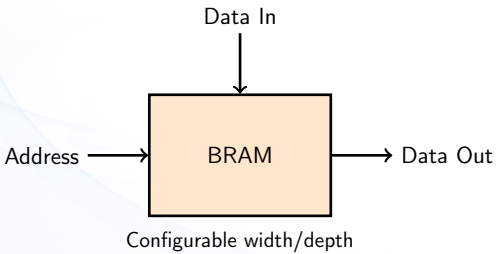
- A Look-Up Table (LUT) is a fundamental building block in FPGAs that implements combinational logic.
- It can be configured to produce any output for a given set of inputs, effectively acting as a small truth table.



2-input LUT

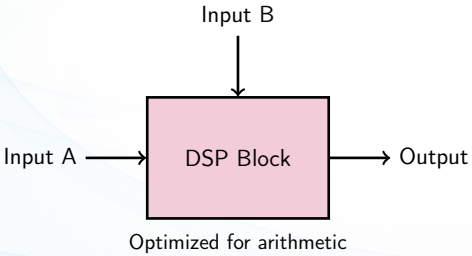
BRAM

- Block RAM (BRAM) is a type of on-chip memory in FPGAs that provides fast access to data.
- It is organized in blocks and can be configured to support various data widths and depths, making it suitable for storing large datasets or implementing FIFOs.



DSP Block

- A DSP (Digital Signal Processing) block is a specialized hardware unit in FPGAs designed to efficiently perform arithmetic operations, such as multiplication and addition, which are common in signal processing applications.
- DSP blocks can be configured to support various data widths and can be used to implement complex mathematical functions, making them essential for high-performance computing tasks.



Core concepts

Pipelining

Pipelining is a technique that divides a computational task into multiple stages, allowing multiple operations to be processed simultaneously.

Key Concepts:

- Each stage processes a different data item
- New input accepted every clock cycle
- Dramatically improves throughput
- Latency = number of pipeline stages
- Initiation Interval (II) = cycles between accepting new inputs

Example: 3-Stage Pipeline

- Stage 1: Read data
- Stage 2: Compute
- Stage 3: Write result

Without pipeline: 3 cycles/operation
With pipeline: 1 output/cycle (after initial fill)

Trade-off

Pipelining increases throughput but also increases latency and resource usage.

Comparison: Architecture

FPGA vs GPU vs CPU

CPU:

- Fixed sequential architecture
- Few cores (2-128)
- High clock frequency (GHz)
- Von Neumann architecture
- General-purpose instruction set

GPU:

- Fixed SIMD architecture
- Thousands of simple cores
- Optimized for data parallelism
- High memory bandwidth
- Thread-based execution

FPGA:

- Reconfigurable fabric
- Thousands of logic elements
- Custom datapath for each task
- Lower clock frequency (MHz)
- Spatial computing

Key Difference

CPUs execute instructions sequentially; FPGAs implement custom hardware circuits for parallel execution.

Comparison: Parallelism

FPGA vs GPU vs CPU

CPU:

- Task-level parallelism (multi-core)
- Instruction-level parallelism (ILP)
- SIMD vector instructions
- Thread-based concurrency
- Limited by core count

GPU:

- Massive data parallelism
- SIMT (Single Instruction, Multiple Threads)
- Thousands of threads
- Best for uniform workloads

FPGA:

- Massive fine-grained parallelism
- Pipeline parallelism
- Custom parallel datapaths
- Hardware-level concurrency
- Spatial execution

Performance

FPGAs can execute hundreds of operations simultaneously in a single clock cycle through custom parallel circuits. GPUs offer higher peak throughput for regular data-parallel tasks; FPGAs excel at irregular, control-intensive algorithms.

Comparison: Memory

FPGA vs GPU vs CPU

CPU:

- Cache hierarchy (L1/L2/L3)
- Unified memory space
- High-capacity DRAM
- Cache coherency protocols
- Virtual memory support
- Limited bandwidth per core

GPU:

- High-bandwidth
- Shared memory per block
- Global memory
- Large capacity (GBs)
- Memory bandwidth bound

FPGA:

- Distributed on-chip memory
- Custom memory hierarchy
- Block RAM and registers
- Direct memory access
- Multiple independent ports

Advantage

FPGAs can create custom memory subsystems with optimal access patterns for specific algorithms, eliminating cache misses.

Programming an FPGA

HDL programming, while powerful, can be complex and time-consuming with these devices, which is where High-Level Synthesis (HLS) comes into play.

HLS allows developers to write high-level code (in C/C++) that describes the desired behavior of the hardware, and then automatically generates the corresponding HDL code. This approach abstracts away much of the low-level details of hardware design, making it more accessible to software developers and enabling faster development cycles.

Main goal of HLS

To attract software developers to design and implement hardware accelerators on FPGAs without needing to learn traditional HDL languages.

What is High-Level Synthesis?

- High-Level Synthesis (HLS) is a design process that takes high-level descriptions of algorithms and generates hardware implementations.
- It uses a subset of C/C++ (or other high-level languages) to describe the functionality, and then applies synthesis techniques to create optimized hardware designs.

Differences in Programming Model

The programming model for FPGAs using HLS differs significantly from traditional CPU and GPU programming:

- A CPU/GPU program is a sequence of operations to be executed in a core/array of cores, i.e. the program extends in time.
- In an FPGA a program defines what fixed operation each component does (at all times), i.e. it extends in space in the FPGA.

HLS Language Basics

HLS uses a subset of C/C++ to describe hardware behavior.

Key constructs include:

- Functions: represent hardware modules
- Simple Data types: integers, floating-point, fixed-point, and custom types for hardware optimization
- Arrays: fixed-size arrays (no dynamic memory allocation)
- Structured data types: structs for grouping related data
- Control structures: if-else, switch-case for decision-making
- Loops: can be unrolled or pipelined for parallelism

HLS Language

Functions 1

In HLS, functions are used to **define hardware modules**. Each function corresponds to a block of hardware that performs a specific operation. They can be **instantiated multiple times** to create parallel hardware structures.
Functions can be called from other functions, allowing for hierarchical design and modularity.

They may have **input and output parameters** that represent the data flowing into and out of the hardware module. These parameters can be passed by value or by reference, depending on the desired hardware behavior.

They may also have **internal states** (e.g., static variables) that allow them to maintain information across function calls, which is essential for implementing sequential logic and state machines.

HLS Language

Functions 2

Functions can be "annotated" with HLS-specific directives (pragmas) to guide the synthesis process and optimize the resulting hardware.

- For example, a function can be marked as **inline** to suggest that it should be expanded at the call site.
- Functions can be **pipelined** to improve performance. Pipelining allows multiple operations to be processed simultaneously, increasing throughput while managing latency.

The behavior of the hardware is determined by the logic within these functions, and how they are connected together. A specific function as to be defined as the **top-level function**, which serves as the entry point for the synthesis process and represents the main hardware module that will be generated.

HLS Language

Data types

HLS supports a variety of data types that are essential for describing hardware behavior:

- **Simple data types:** These include standard C/C++ types such as integers (e.g., int, short) and floating-point numbers (e.g., float, double). HLS also provides fixed-point data types that allow for precise control over the number of bits used for the integer and fractional parts, which is crucial for optimizing hardware performance and resource usage.
- **Structured data types:** HLS supports structs, which allow developers to group related data together. This is useful for representing complex data structures in hardware. However, dynamic memory allocation (e.g., using pointers) is not supported in HLS, so all data structures must have a fixed size known at compile time.

HLS Language

Integers

ap_[u]int<N>: unsigned/signed integer with N bits (N=1-512)

- all standard integer operations are supported
- bit manipulation
  - single bit access: x[7] extracts the 7th bit of x
  - bit concatenation: (x,y) combines x and y into a wider integer
  - slicing: x.range(7,0) extracts bits 7 to 0 of x
- vitis automatically determines number of bits needed for operations (the sum of two 8-bit integers is a 9-bit integer).

Integer can be used for most of the math with attention to overflow and precision loss.

$x_i = int\left(x_f \cdot \frac{1}{LSB}\right)$ Converting a floating-point number to an integer representation by scaling it according to the least significant bit (LSB)

$c = (a \cdot b) >> n$ Performing multiplication and then right-shifting the result to adjust for the scaling

Vitis HLS

Vitis HLS is a high-level synthesis tool from Xilinx that allows developers to design hardware using C, C++, and OpenCL.

It provides a comprehensive environment for writing, simulating, and synthesizing hardware designs, making it easier to create efficient FPGA implementations.

We will be using Vitis HLS 2023.2 for our hands-on projects. Even though newer versions of Vitis change the GUI and the workflow, the core concepts and design principles remain consistent across versions.

Vitis

Project structure

A Vitis HLS project typically consists of several key components, tipically included in a **TCL file** that describes the project configuration and build process. These components include:

- **Source files:** C/C++ files containing the high-level code that describes the hardware behavior.
- **Testbench:** A separate C/C++ file that simulates the hardware design and verifies its functionality.
- **Configuration directives:** Several directives that specify synthesis options, target device, and other parameters for the design process.

Vitis

Design flow

The design flow in Vitis HLS typically involves the following steps:

- **Writing the high-level code:** Develop the C/C++ code that describes the desired hardware functionality. **Write the testbench** as well.
- **Simulating the design:** Use the testbench to simulate the design and verify its correctness before synthesis. **Two types of simulation**
- **Synthesizing the design:** Run the synthesis process to generate the corresponding HDL code and hardware implementation. The HDL code can be further processed to create a bitstream for programming the FPGA.
- **Analyzing results:** Review synthesis reports, resource utilization, and performance metrics to evaluate the design.

Vitis

Commands

Vitis HLS can execute the following commands:

- **csim**: C/C++ simulation of the design using the testbench to verify functional correctness.
- **cosim**: Co-simulation that combines C simulation with HDL simulation to verify the design at a lower level of abstraction.
- **syn**: Synthesis of the high-level code into HDL, generating the hardware implementation and associated reports.
- **export**: Export the synthesized design for use in other tools (e.g., Vivado) to create a bitstream for FPGA programming.

Vitis

Open a project and Makefile

Run a Vitis HLS project from the command line interface via a TCL file:

```
vitis_hls -f <TCL file>
```

- to create a project from the file
- to run the automatic build process (simulation, synthesis, export)

Open a Vitis HLS project in the GUI:

```
vitis_hls -p <project directory>
```

- to browse report files
- to use as and IDE for writing code and running individual steps of the design flow

For ease of use, we will be using a Makefile to automate the build process for our HLS projects.

The Makefile will include targets for simulating, synthesizing, and exporting the design, as well as cleaning up generated files.

Hands-on structure

Recap

- Each Hands-on project will have a specific name and a clear set of objectives.
- We will start with simple designs and gradually increase complexity as we progress through the course
- Each project will have its own directory within the course repository, containing all necessary files and instructions for completion.
- Each project will include a checklist of tasks to be completed

Github Repository

git clone https://github.com/mmirko/FPGALab.git

Hands-on

simple_sum

```
git clone https://github.com/mmirko/FPGALab.git ; cd FPGALab/simple_sum
```

- **Description:** A simple HLS design that implements a basic arithmetic addition of two integers.
- **Objective:** Acquire familiarity with the Vitis HLS tool, understand the HLS design flow, and learn how to write a simple C++ function that can be synthesized into hardware.
- **Objective:** Synthesize the design to generate the corresponding hardware implementation and analyze the results.

Hands-on

simple_mult

```
git clone https://github.com/mmirko/FPGALab.git ; cd FPGALab/simple_mult
```

- **Description:** A simple HLS design that implements a basic arithmetic multiplication of two integers.
- **Objective:** Synthesize the design to generate the corresponding hardware implementation and analyze the results.

Hands-on

simple_div

```
git clone https://github.com/mmirko/FPGALab.git ; cd FPGALab/simple_div
```

- **Description:** A simple HLS design that implements a basic arithmetic division of two integers.
- **Objective:** Synthesize the design to generate the corresponding hardware implementation and analyze the results.

HLS Language

Floating-point

HLS supports standard floating-point types (float, double) that conform to the IEEE 754 standard.

- Floating-point operations are more resource-intensive than fixed-point or integer operations, so they should be used judiciously in hardware design.
- HLS provides directives to optimize floating-point computations, such as using reduced precision or implementing custom floating-point formats.

Use floating-point when precision is critical and resource usage is acceptable.

For example, in scientific computing or when dealing with a wide dynamic range of values.

Hands-on

floating_point

```
git clone https://github.com/mmirko/FPGALab.git ; cd FPGALab/floating_point
```

- **Description:** A simple HLS design that implements floating-point arithmetic operations.
- **Objective:** Learn how to create new functions and integrate them into a project.
- **Objective:** Synthesize several floating-point operations and analyze the results in terms of resource usage and fill in the following table.

Hands-on

floating_point

Operation	LUTs	FFs	DSPs	Latency (cycl)	II (cycl)
Float Addition (no pipeline)	241	325	2	6	7
Float Multiplication (no pipeline)	104	147	3	3	4
Float Division (no pipeline)	65	12	0	11	12
Double Addition (no pipeline)					
Double Multiplication (no pipeline)	168	410	8	6	7
Double Division (no pipeline)	148	31	0	30	31

HLS Language

Fixed-point

HLS provides fixed-point data types (ap_fixed) that allow for precise control over the number of bits used for the integer and fractional parts of a number.
ap_[u]fixed<W,I>: unsigned/signed fixed-point with W total bits and I integer bits (W=1-512, I=0-W)

- Fixed-point arithmetic is more resource-efficient than floating-point, making it ideal for performance-critical applications where precision can be traded off for speed and area.
- HLS supports a wide range of fixed-point operations, including addition, subtraction, multiplication, and division, as well as bit manipulation and scaling.
- When using fixed-point types, it is important to carefully manage precision and avoid overflow, especially during multiplication and accumulation operations.

Use fixed-point when you need a balance between precision and resource usage.

For example, in digital signal processing or machine learning inference where the dynamic range is limited and performance is critical.

Hands-on

fixed_point

```
git clone https://github.com/mmirko/FPGALab.git ; cd FPGALab/fixed_point
```

- **Description:** A simple HLS design that implements fixed-point arithmetic operations.
- **Objective:** Synthesize several fixed-point operations and analyze the results in terms of resource usage.

Hands-on

intricate

```
git clone https://github.com/mmirko/FPGALab.git ; cd FPGALab/intricate
```

- **Description:** A more intricate HLS design that implements an arithmetic operation using several functions called in a nested manner.
- **Objective:** Familiarize with concepts such as latency, throughput, and resource utilization in the context of HLS designs.

Hands-on

intricate_pipelined

```
git clone https://github.com/mmirko/FPGALab.git ; cd FPGALab/intricate_pipelined
```

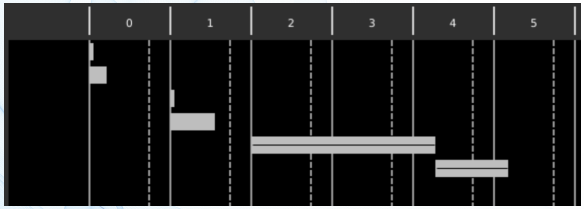
- **Description:** A more intricate HLS design that implements an arithmetic operation using several functions called in a nested manner.
- **Objective:** Learn how to apply pipelining techniques to improve the performance of an HLS design and analyze the impact on latency and throughput.

Hands-on

intricate comparison

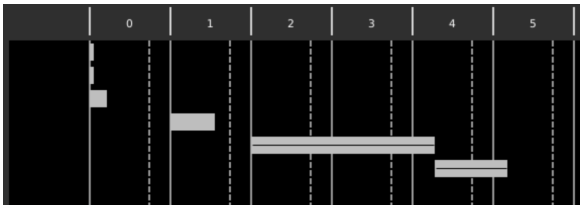
Without pipelining:

LUTs	FFs	DSPs	Lat (cy)	II (cy)
65	38	2	5	6



With pipelining:

LUTs	FFs	DSPs	Lat (cy)	II (cy)
62	150	2	5	1



Hands-on

floating_point_pipelined

```
git clone https://github.com/mmirko/FPGALab.git ; cd FPGALab/floating_point_pipelined
```

- **Description:** An example of HLS functions that use floating-point operations.
- **Objective:** Create a whole project from scratch, including the testbench, and learn how to apply pipelining techniques to improve the performance of an HLS design.

HLS Language

arrays and structs

HLS supports fixed-size arrays that can be used to represent collections of data in hardware. The syntax for declaring an array is similar to standard C/C++ arrays.

- Arrays can be mapped to FFs, BRAMs or FIFOs depending on their size and access patterns.
- HLS provides directives to control how arrays are implemented in hardware, such as specifying the memory type or partitioning the array for parallel access.

HLS also supports structs and small classes, which allow developers to group related data together.

Hands-on

arrays

```
git clone https://github.com/mmirko/FPGALab.git ; cd FPGALab/arrays
```

- **Description:** An examples of HLS functions that use arrays with different access patterns.
- **Objective:** Understand how to use arrays in HLS designs and analyze how different access patterns affect resource usage and performance.

HLS Language

Control structures

HLS supports standard control structures such as if-else statements, switch-case statements, and loops (for, while, do-while).

- Control structures are used to implement decision-making and iterative behavior in hardware designs.
- HLS provides directives to optimize control structures for hardware implementation, such as unrolling loops or pipelining them for improved performance.

HLS Language

Branching

Branching in HLS is implemented using if-else statements and switch-case statements.

- When synthesizing branching logic, HLS generates all possible paths through the control structure.
- Latency will be the longest path through the control structure.
- Resource will be the sum of all paths through the control structure.

Hands-on

branching

```
git clone https://github.com/mmirko/FPGALab.git ; cd FPGALab/branching
```

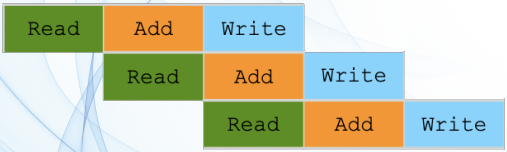
- **Description:** An example of HLS functions that use branching control structures.
- **Objective:** Understand how to use branching control structures in HLS designs and analyze how they affect resource usage and performance.

HLS Language

Loops

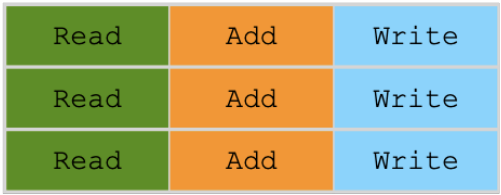
Loops in HLS can be implemented using for, while, and do-while constructs. the syntax is similar to standard C/C++ loops.

```
for(i = 0; i < 3; i++)
    a[i] = a[i] + 1;
```



Pipelined

Sequential



Unrolled

Hands-on

Loops

```
git clone https://github.com/mmirko/FPGALab.git ; cd FPGALab/loops
```

- **Description:** An example of HLS functions that use loops.
- **Objective:** Understand how to use loops in HLS designs and analyze how different loop optimizations (pipelining, unrolling) affect resource usage and performance.

Using Accelerators

- Once we have designed and synthesized our hardware accelerator using HLS, the next step is to integrate it into a larger system and use it for computation.
- This typically involves interfacing the accelerator with a processor (e.g., ARM CPU) and managing data transfer between the processor and the accelerator.
- We will explore how to use the generated hardware accelerators in a real system, including how to write software that interacts with the accelerator.

Using Accelerators

xrt and platform

To use the generated hardware accelerators, we will leverage the Xilinx Runtime (XRT) and the platform provided by Xilinx.

- XRT (Xilinx Runtime) is a software library that provides an API for managing and interacting with hardware accelerators on Xilinx FPGAs.
- The platform describes the hardware configuration of the FPGA, including the available resources, memory interfaces, and communication protocols.

Using Accelerators

C++ and OpenCL

To interact with the hardware accelerators in C/C++, we have two main options:

- Using the XRT API directly to manage the accelerator and data transfer.
- Using OpenCL, which provides a higher-level abstraction for programming heterogeneous platforms, including FPGAs.

```
auto device = xrt::device(0);
auto xclbin = device.load_xclbin("firmware.xclbin");
auto kernel = xrt::kernel(device, xclbin, "sum");
auto run = kernel(10,20);
run.wait();
```

Interfacing the Accelerator

AXI

- *AXI (Advanced eXtensible Interface)* is a communication protocol developed by ARM as part of the AMBA (Advanced Microcontroller Bus Architecture) specification
- Industry-standard interface for connecting IP blocks in SoC designs
- Widely used in FPGA designs, especially with Xilinx devices
- Key benefits:
  - Standardized communication between hardware modules
  - Supports high-performance, high-bandwidth operations
  - Enables reusability and interoperability of IP cores
  - Facilitates integration with ARM processors and other AMBA components

Use in FPGA HLS

AXI is the primary interface for connecting custom accelerators to processors and memory in FPGA-based SoC designs.

AXI Interface Types

AXI comes in three variants, each optimized for different use cases:

AXI4 (Full):

- High-performance memory-mapped interface
- Supports burst transactions
- Multiple outstanding addresses
- Used for memory access (DDR, BRAM)
- Complex but flexible

AXI4-Lite:

- Simplified memory-mapped interface
- Single transaction only
- Lower resource usage
- Used for control-status registers
- Easy to implement

AXI4-Stream:

- Point-to-point streaming interface
- Unlimited data burst size
- No addresses (unidirectional flow)
- Used for data streaming
- Optimal for pipelining

Pynq

- Pynq is an open-source project from Xilinx that provides a Python-based interface for programming and controlling FPGAs.
- It allows users to interact with FPGA hardware using Python, making it easier to develop and test FPGA applications without needing to write low-level HDL code.

Overlays

- An overlay is a pre-designed hardware configuration that can be loaded onto an FPGA to provide specific functionality.
- Overlays can be thought of as "hardware libraries" that offer ready-to-use components and interfaces, allowing developers to quickly implement complex applications without needing to design the hardware from scratch.

Hands-on

bulk_sum

```
git clone https://github.com/mmirko/FPGALab.git ; cd FPGALab/bulk_sum
```

- **Description:** A simple application that performs a bulk sum operation on two arrays of predefined size and returns the result array of the sums.
- **Objective:** See how to extend the HLS design to build an real accelerator and use it on a Pynq framework.
- **Objective:** Learn how to use overlays to load the accelerator onto the FPGA and interact with it using Python.

Thanks!

Thank you for your attention!

Slides and content partially adapted from the following sources:
G.Petrucciani, 2023 Course on HLS
S.Summers, 2025 Introductory course to VHDL and HLS FPGA programming