Funded by
the European Union

Joint Education for Advanced Chip Design in Europe (Edu4Chip)

Course Material for Tape-out Courses

Deliverable Report

Author(s):	Martin Schoeberl, Luca Pezzarossa (DTU) Dovydas Liutkus (DTU)[FPGA Verification] Alexandre Menu, Jean-Max Dutertre (IMT) Nooshin Nosrati, Ahmed Hemani (KTH) Friedrich Racz (Syosil) Matti Käyrä, Arto Oinonen (TAU) Paul Genssler, Michael Pehl (TUM)
Version:	0.1
Dissemination Level:	Public
Work Package No:	4
Deliverable Number:	D 4.2
Lead Beneficiary:	IMT

CONTENTS:

CHAPTER

ONE

INTRODUCTION

This document provides the course materials for the Edu4Chip ASIC chip design labs. It covers the complete design flow from RTL description to physical tapeout, organized as a sequence of self-contained lab chapters each targeting one step of the flow. The Didactic SoC serves as the integration platform throughout the labs: students design, verify, and implement a custom hardware subsystem that is incrementally integrated into the chip. The design flow progresses across chapters as follows:

- **Chapter 2** — Didactic SoC platform architecture, memory map, peripherals, and boot sequence
- **Chapter 3** — SoC-level RTL simulation and testbench environment
- **Chapter 4** — Student subsystem front-end functional verification
- **Chapter 5** — Hardware/software co-design and firmware development
- **Chapter 6** — FPGA-based pre-silicon validation
- **Chapter 7** — Logic synthesis and optimization
- **Chapter 8** — Subsystem physical layout: floorplan, place and route, sign-off
- **Chapter 9** — Chip-level layout and subsystem integration
- **Chapter 10** — IC validation and sign-off
- **Chapter 11** — Open-source EDA tools

1.1 The Edu4Chip project

The Edu4Chip partner universities, the Technical University of Denmark, the KTH Royal Institute of Technology, the Ecole des Mines de Saint-Etienne of IMT, Tampere University, and the Technical University of Munich, aim to design and implement harmonized study programs in ASIC chip design. In these programs, students are expected to carry out a complete chip design, from the high-level description of the hardware to the manufacturing and testing of an ASIC.

1.2 Who is this for?

This document serves as a reference for academics, students, motivated individuals and open source enthusiasts wishing to use the Didactic SoC platform to teach and learn ASIC chip design. Academics can find lab materials that can be readily used with the Didactic SoC or adapted in chip design courses. Students can refer to this document during a lab assignment to get guidance and background knowledge on each step of the design flow, as well as specific instructions and documentation related to the Didacic SoC platform. Motivated individuals and open source enthusiasts can benefit from a comprehensive course on chip design illustrated with practical examples on the Didactic SoC using open-source EDA tools.

1.3 Organisation of this document

The organisation of this document follows a didactic purpose, with each chapter reflecting a step of the design flow and a lab targeting this specific step. As such, it does not reflect the design methodology used by the partner universities to design an extensible didactic integrated circuit. It does not reflect either the steps of an industrial design methodology in which the application drives the physical specifications of the circuit. As each chapter is meant to be given as guidance materials in a chip design lab, chapters may also contain redundant information to support standalone use of each chapter in lab sessions.

1.4 Organisation of the Didactic SoC repository

The Didactic SoC repository[1] is organized in several folders which separate design artifacts. The top-level directory structure is as follows:

```
<ROOT_DIR>/
├── build/                # generated by make, not in repo
├── fpga/
├── ipxact/
├── scripts/
├── sim/
├── src/
├── sw/
├── syn/
└── verification/
```

The descriptions of these folders are given below:

- `build/` — All tool outputs generated by make commands.
- `fpga/` — Tool scripts, constraints and documentation for FPGA target platforms.
- `ipxact/` — IP-XACT XML definition files.
- `scripts/` — General-purpose scripts not belonging to any other specific category.
- `sim/` — Simulation Makefiles and scripts, including waveform and supporting files. Targets Questa and Verilator via Bender integration.
- `src/` — RTL source files organized in subfolders. Student subsystems can be added as submodules, Bender dependencies, or directly.
- `sw/` — Baremetal software, common headers and RISC-V test cases, including subsystem-specific tests.
- `syn/` — Synthesis Makefiles and commands using open-source tools only.
- `verification/` — Verilator setup and PyUVM verification platform for student subsystems.

[1] https://github.com/Edu4Chip/Didactic-SoC

CHAPTER

TWO

THE DIDACTIC SOC PLATFORM[6]

The Didactic SoC platform[1] is the baseline SoC template for Edu4Chip. It offers a microcontroller-scale open-source platform for education. The SoC was developed based on principles of simplicity, extendability, reusability, and ease of integration.

2.1 Specification

The Didactic SoC architecture was designed around two distinct functional sections: the management section, also called the staff section, and the student sections in which student subsystems are integrated. These sections are highlighted in Fig. 2.1: the management section on the left of the figure is connected through a global interconnect network (ICN) to the student subsystems (SS) on the right of the figure. Student sections integrate custom subsystems with a well-defined interface that can be individually selected. The generic interface of the subsystems is described in *Student subsystems*.

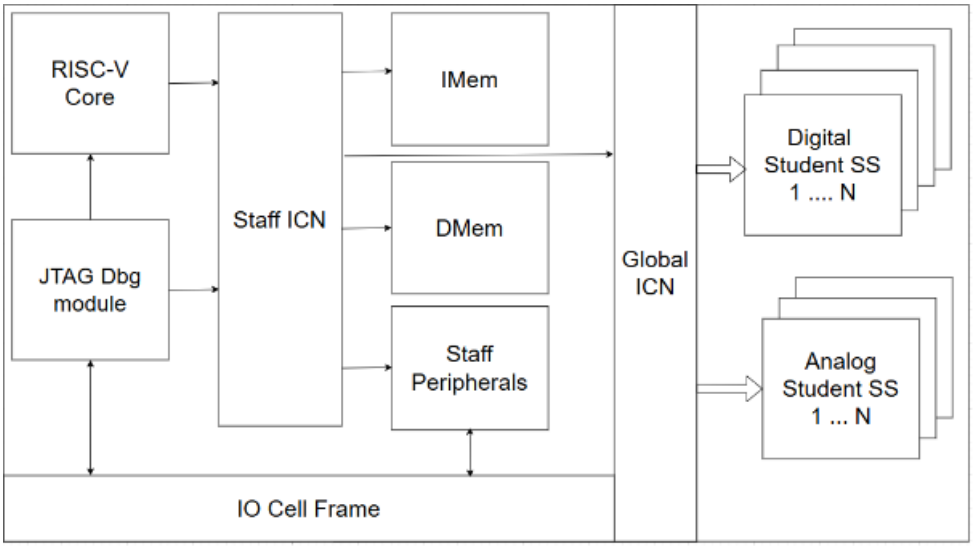

Fig. 2.1: Block diagram of the Didactic SoC architecture.

The staff section of the chip offers:

- a RISC-V Ibex[4] processor core that implements the RV32IMC instruction set architecture
- a debug module accessible over JTAG from the PULP open source project[3] which is compliant with the RISC-V Debug Specification

[6] AI tools in combination with careful proofereading and rewriting were partially used to support the writing of this section.
[1] https://github.com/Edu4Chip/Didactic-SoC
[4] https://github.com/lowRISC/ibex
[3] https://github.com/pulp-platform

- one 16 KiB SRAM instruction memory (`IMEM`) and one 16 KiB SRAM data memory (`DMEM`)
- a set of external peripherals (UART, SPI) from the PULP project[Page 3, 3] to communicate with the SoC
- a peripheral interface to up to five freely customisable submodules
- dedicated circuitry that controls subsystem reset and clock enable on one hand, and configures the connection of the subsystems to the pads of the SoC on the other

The initial specification targets the GF 22 nm ASIC technology from Europractice with a maximum frequency of 100 MHz and a maximum area of 2.5 mm^2 for both the staff section and the student subsystems. The system provides 16 GPIO connections and SPI and UART peripherals. The core and IO voltages are determined by the technology node: 0.8 V for the core and 1.2 V / 1.5 V / 1.8 V for the IOs, which requires the test PCB to use level shifters compatible with 3.3 V peripheral modules.

The above specifications are summarized in Table 2.1.

Table 2.1: Didactic SoC technical specifications

Parameter	Value
Technology node	GF 22nm FDX (Europractice)
Maximum frequency	100 MHz
Maximum area	2.5 mm^2 (staff section + student subsystems)
Core voltage	0.8 V
IO voltage	1.2 V / 1.5 V / 1.8 V
CPU core	RISC-V Ibex, RV32IMC ISA
Instruction memory	16 KiB SRAM
Data memory	16 KiB SRAM
Debug interface	JTAG (RISC-V Debug Specification)
GPIO	16 pins
Serial interfaces	UART, SPI
Student subsystem slots	Up to 5
Student subsystem bus	APB

2.1.1 Pinout

The students IPs and peripherals of the Didactic SoC require 32 IO pins. On top of these, analog student subsystems can have their own IO pins. The digital IO connections are routed through a centralized module to handle instantiation of technology specific cells. The rest of the SoC IO area is filled with power and ground connections during the physical implementation stage.

2.1.2 IP-XACT model

IP-XACT is an IP exchange format implemented using XML. It is specified in the IEEE-1685 standard and managed by Accelera. This format enables designers to capture structural descriptions, memory content and information pertaining to design files and documentation. The XML description allows in turn to describe design information in a standardized, tool-friendly format. Didactic platform uses the Kactus2 Graphical User Interface to represent and manipulate the IP-XACT description of the SoC.

2.1.3 Dependency management

This SoC depends on open-source implementations provided by developers at various repositories, primarily on GitHub under the OpenHW Group[2] and PULP Platform[Page 3, 3]. These reused hardware modules reference specific commits or tags to track the version of each dependency that corresponds to a given release of the Didactic SoC. Both the Didactic SoC and PULP Platform modules depend on the same bus-specific modules, which leads to the traditional dependency management challenge in hardware development.

This challenge can be addressed in various ways, such as monolithic repositories that copy dependencies inline, or through the use of git submodules that always point to a specific version of a repository. The former makes it difficult to update modules while crediting their authors, and the latter leads to inflexible repository structures. During module development this is not usually a roadblock, but whenever an additional developer joins the project, it tends to cause issues when building consistent development environments across partners.

To facilitate managing repositories at multiple levels in hardware projects, the PULP Platform developed a tool called Bender[5]. It uses a YAML configuration file to represent project dependencies and enforce a specific project structure for submodules. It is then able to resolve dependency chains of all submodules that use the same structure. This mechanism allows the top-level project to easily fetch third-party IPs and manage the versions of the submodules.

2.2 Memory and bus architecture

2.2.1 Bus architecture

The Didactic SoC bus architecture was designed to be both topology- and type-agnostic. The only requirement is that the bus type must be addressable to expose peripherals and subsystems through a unified memory map. IP-XACT modules were used to keep track of bus types and memory-mapped components.

The SoC was first developed using the AXI4LITE ARM AMBA protocol for high-speed bus communication, before switching to the open-source OBI protocol standardised by the OpenHW Group for the taped-out version of the SoC. An OBI-to-APB bridge is used to simplify the interface of the student subsystems.

The SoC bus is currently implemented as two hierarchically separated layers of fully connected crossbars in the global interconnect network (ICN). The first layer routes connections to all submodules of the staff section, and the second layer manages CPU access to the student subsystems.

2.2.2 Memory organisation

System memory map

The memory address layout of the soC is given in Table 2.2. The SoC memory is partitioned across modules, each occupying a fixed address range in the 32-bit address range. The address space was not manually packed.

[2] https://github.com/openhwgroup
[5] https://github.com/pulp-platform/bender

Table 2.2: System memory layout

System	Base Address
Instruction memory	'h01000000
Data memory	'h01010000
Debug module	'h01020000
Peripherals	'h01030000
Control registers	'h01040000
Student subsystems	'h01050000

Peripheral memory map

The memory address layout of the SoC peripherals is given in Table 2.3.

Table 2.3: Peripheral memory layout

Peripheral	Base Address
GPIO	'h01030000
UART	'h01030100
SPI	'h01030200

Student subsystems memory map

The memory address layout of the available slots for student subsystems is given in Table 2.4.

Table 2.4: Student subsystems memory layout

Student subsystem	Base Address
SS0	'h01050000
SS1	'h01060000
SS2	'h01070000
SS3	'h01080000
SS4	'h01090000

2.3 Debug Support

Debug access is one of the standard access modes on a SoC, as an alternative to independent or guided boot. In the Didactic SoC, it is used as the sole boot option for simplicity.

2.3.1 JTAG debug interface

JTAG is an IEEE standard that defines SoC access through a dedicated serial interface. The debug module connected to this interface accepts commands from a host PC. This allows standard tools such as GDB to control the SoC.

2.3.2 Device programming

Device programming is performed in C. The repository includes generic helper functions in header files to support rapid development of test and application code.

2.3.3 Boot sequence

The first iteration of the SoC does not have an independent boot capability. It is fully externally controlled via JTAG. The boot sequence is as follows:

1. SoC is connected to host PC via USB
2. SoC is lifted from reset
3. OpenOCD takes control of debug module via JTAG through ftdi-module

A) Manual operation through commands:

4. Connection is established through RISC-V gdb
5. RISC-V gdb is used to control SoC with direct commands

B) Programming flow:

4. Baremetal program is compiled using RISC-V toolchain
5. RISC-V gdb is used to control the SoC and preload instruction memory
6. RISC-V gdb lifts core from halt
7. Core reads instruction memory and starts program execution

Later iterations of the Didactic SoC may include an independent boot mode, in which the core would start from a program stored in ROM.

2.4 Controller

The controller block (`SS_Ctrl_reg_array`) is the central control register bank mapped at base address `0x01040000`. It manages CPU fetch enable, subsystem reset and clock gating, PMOD routing, and IO cell pad configuration. The complete register map is given in Table 2.5.

Table 2.5: Controller register map (base address `0x01040000`)

Register name	Offset	Description
`fetch_en`	`0x000`	Ibex CPU instruction-fetch enable
`ss_rst`	`0x004`	Subsystem and ICN reset control (active-high)
`icn_ss_ctrl`	`0x008`	ICN subsystem control word
`ss_0_ctrl`	`0x00C`	Subsystem 0 clock and IRQ control
`ss_1_ctrl`	`0x010`	Subsystem 1 clock and IRQ control
`ss_2_ctrl`	`0x014`	Subsystem 2 clock and IRQ control
`ss_3_ctrl`	`0x018`	Subsystem 3 clock and IRQ control
`ss_4_ctrl`	`0x01C`	Subsystem 4 clock and IRQ control
`ss_ctrl_reserved_1`	`0x020`	Reserved for a future subsystem slot
`pmod_sel`	`0x024`	PMOD connector host-selection mux
`io_cell_cfg_0`	`0x028`	IO cell configuration for the UART RX pad
`io_cell_cfg_1`	`0x02C`	IO cell configuration for the UART TX pad
`io_cell_cfg_2`	`0x030`	IO cell configuration for the SPI SCK pad
`io_cell_cfg_3`	`0x034`	IO cell configuration for the SPI CSN0 pad
`io_cell_cfg_4`	`0x038`	IO cell configuration for the SPI CSN1 pad
`io_cell_cfg_5`	`0x03C`	IO cell configuration for the SPI DATA0 pad
`io_cell_cfg_6`	`0x040`	IO cell configuration for the SPI DATA1 pad
`io_cell_cfg_7`	`0x044`	IO cell configuration for the SPI DATA2 pad
`io_cell_cfg_8`	`0x048`	IO cell configuration for the SPI DATA3 pad
`io_cell_cfg_9` – `io_cell_cfg_24`	`0x04C` – `0x088`	IO cell configuration for GPIO[0]–GPIO[15] pads
`return_reg_0`	`0x100`	Post-main idle-loop instruction word 0
`return_reg_1`	`0x104`	Post-main idle-loop instruction word 1
`boot_reg_0`	`0x180`	Boot idle-loop instruction word 0
`boot_reg_1`	`0x184`	Boot idle-loop instruction word 1

The register fields are described below.

`fetch_en` (offset `0x000`)

- `fetch_en_reg` [3:0] — Ibex instruction-fetch enable. Reset value `0x5`. Write a non-zero value to enable instruction fetch; write `0x0` to halt the CPU.

`ss_rst` (offset `0x004`)

- `icn_rst` [0] — ICN reset. Write 1 to hold the interconnect in reset; write `0` to release it.
- `ss_0_rst` [1] — Subsystem 0 reset. Write 1 to hold SS0 in reset; write `0` to release.
- `ss_1_rst` [2] — Subsystem 1 reset. Write 1 to hold SS1 in reset.
- `ss_2_rst` [3] — Subsystem 2 reset. Write 1 to hold SS2 in reset.
- `ss_3_rst` [4] — Subsystem 3 reset. Write 1 to hold SS3 in reset.
- [31:5] — Reserved, write `0`.

`icn_ss_ctrl` (offset `0x008`)

- `icn_ctrl` [30:0] — Control word forwarded to the ICN subsystem port. Currently unused by the ICN logic.

`ss_N_ctrl` (offsets `0x00C`–`0x01C`, one register per subsystem N = 0–4)

- `ssN_clk_en` [0] — Standard clock enable. Write 1 to gate the clock on for subsystem N.
- `ssN_fast_clk_en` [1] — Fast clock enable. Write 1 to enable the high-speed clock for subsystem N.
- [30:2] — Reserved, write `0`.
- `ssN_irq_en` [31] — IRQ enable. Write 1 to route the subsystem N interrupt to the SoC interrupt controller.

`pmod_sel` (offset `0x024`)

- `pmod_ctrl` [7:0] — PMOD routing select. Write the subsystem index (0–3) whose GPIOs should be routed to the PMOD connector. Any value outside 0–3 routes the staff-section GPIOs instead. Reset value `0x4` (staff-section GPIOs active by default).

`io_cell_cfg_N` (offsets `0x028`–`0x088`, one register per IO pad)

Each register configures one IO pad; only bits [4:0] are connected to the IO cell. Reset value `0x0D` (`5'b01101`).

- `dir` [0] — Pad direction. `0` = output (pad driven from core); 1 = input (pad tristated, value sampled to core).
- [4:1] — Reserved for additional cell parameters (drive strength, slew rate, Schmitt trigger). Not connected in the simulation model.

`return_reg_0` / `return_reg_1` (offsets `0x100` / `0x104`)

- [31:0] — Two consecutive instruction words executed at the CPU program counter after `main()` returns. Both reset to `0x6F` (RISC-V `JAL x0, 0`), so the CPU enters a safe self-loop after program completion.

`boot_reg_0` / `boot_reg_1` (offsets `0x180` / `0x184`)

- [31:0] — Two consecutive instruction words executed at boot before firmware is loaded over JTAG. Both reset to `0x6F` (`JAL x0, 0`), keeping the CPU in a safe idle state until the debugger preloads instruction memory and releases the core.

2.5 Peripherals

Peripheral modules are standard interface modules that interact with external devices. They are reused from PULP Platform modules and are accessible at fixed hardware addresses.

2.5.1 GPIO

GPIOs are general-purpose IO signals that serve no dedicated function. They can be used for a wide variety of purposes, such as reading sensor values or driving an LED. They can interface with any external module that operates at relatively low speed. The GPIOs are exposed on a PMOD connector, enabling connection to standard PMOD modules such as memories or audio devices. Student subsystems that use these pins must implement their own behaviour and synchronization logic internally. The Didactic SoC instantiates 16 GPIO pins (GPIO[15:0]); GPIO pad direction is controlled via the `io_cell_cfg` registers in the controller block.

Table 2.6: GPIO register map (base address `0x01030000`)

Register name	Offset	Description
REG_PADDIR_00_31	0x00	Pad direction for GPIO[31:0]: 1 = output, 0 = input
REG_GPIOEN_00_31	0x04	GPIO function enable for GPIO[31:0]: 1 enables the GPIO peripheral on that pad
REG_PADIN_00_31	0x08	Sampled input values for GPIO[31:0] (read-only)
REG_PADOUT_00_31	0x0C	Output values for GPIO[31:0]
REG_PADOUTSET_00_31	0x10	Atomic set for GPIO[31:0]: write 1 to drive a pin high; 0 bits unchanged
REG_PADOUTCLR_00_31	0x14	Atomic clear for GPIO[31:0]: write 1 to drive a pin low; 0 bits unchanged
REG_INTEN_00_31	0x18	Interrupt enable for GPIO[31:0]: 1 enables edge detection on that pin
REG_INTTYPE_00_15	0x1C	Interrupt trigger type for GPIO[15:0]: 2 bits per pin
REG_INTTYPE_16_31	0x20	Interrupt trigger type for GPIO[31:16]: 2 bits per pin
REG_INTSTATUS_00_31	0x24	Interrupt status for GPIO[31:0]: set on trigger event, cleared on register read
REG_PADCFG_00_07	0x28	Pad configuration for GPIO[7:0]: 4 bits per pin
REG_PADCFG_08_15	0x2C	Pad configuration for GPIO[15:8]: 4 bits per pin
REG_PADCFG_16_23	0x30	Pad configuration for GPIO[23:16]: 4 bits per pin
REG_PADCFG_24_31	0x34	Pad configuration for GPIO[31:24]: 4 bits per pin
REG_PADDIR_32_63	0x38	Pad direction for GPIO[63:32]
REG_GPIOEN_32_63	0x3C	GPIO function enable for GPIO[63:32]
REG_PADIN_32_63	0x40	Sampled input values for GPIO[63:32] (read-only)
REG_PADOUT_32_63	0x44	Output values for GPIO[63:32]
REG_PADOUTSET_32_63	0x48	Atomic set for GPIO[63:32]
REG_PADOUTCLR_32_63	0x4C	Atomic clear for GPIO[63:32]
REG_INTEN_32_63	0x50	Interrupt enable for GPIO[63:32]
REG_INTTYPE_32_47	0x54	Interrupt trigger type for GPIO[47:32]: 2 bits per pin
REG_INTTYPE_48_63	0x58	Interrupt trigger type for GPIO[63:48]: 2 bits per pin
REG_INTSTATUS_32_63	0x5C	Interrupt status for GPIO[63:32]: set on trigger event, cleared on register read
REG_PADCFG_32_39	0x60	Pad configuration for GPIO[39:32]: 4 bits per pin
REG_PADCFG_40_47	0x64	Pad configuration for GPIO[47:40]: 4 bits per pin
REG_PADCFG_48_55	0x68	Pad configuration for GPIO[55:48]: 4 bits per pin
REG_PADCFG_56_63	0x6C	Pad configuration for GPIO[63:56]: 4 bits per pin

2.5.2 UART

UART is a standardized serial interface typically used to transmit characters to a terminal. The SoC is connected to an FTDI chip which receives data over the UART and forwards it to the host PC over USB. The UART peripheral is a 16450-compatible UART with optional FIFO extensions.

Table 2.7: UART register map (base address `0x01030100`)

Register name	Offset	Width	Description
`RBR_THR_DLL`	`0x00`	8	Receive Buffer / Transmit Holding / Divisor Latch LSB (multiplexed)
`IER_DLM`	`0x04`	8	Interrupt Enable / Divisor Latch MSB (multiplexed)
`IIR_FCR`	`0x08`	8	Interrupt ID Register (read) / FIFO Control Register (write)
`LCR`	`0x0C`	8	Line Control Register
`MCR`	`0x10`	8	Modem Control Register
`LSR`	`0x14`	8	Line Status Register (read-only)
`MSR`	`0x18`	8	Modem Status Register (read-only)
`SCR`	`0x1C`	8	Scratch Register (general-purpose R/W)

2.5.3 SPI

SPI is typically connected to external memories such as an SD card, but can also control any standard SPI device such as sensors. The SPI peripheral supports standard, dual, and quad SPI modes with up to four chip-select lines.

Table 2.8: SPI register map (base address `0x01030200`)

Register name	Offset	Description
`STATUS`	`0x00`	Transfer mode control and chip-select enable (self-clearing command bits)
`CLKDIV`	`0x04`	SPI clock divider
`SPICMD`	`0x08`	SPI command word
`SPIADR`	`0x0C`	SPI address word
`SPILEN`	`0x10`	Transfer lengths: command, address, and data bit counts
`SPIDUM`	`0x14`	Dummy cycle counts for read and write phases
`TXFIFO`	`0x18`	Transmit FIFO (write-only)
`RXFIFO`	`0x20`	Receive FIFO (read-only)
`INTCFG`	`0x24`	Interrupt configuration
`INTSTA`	`0x28`	Interrupt status (read-clear)

2.6 Student subsystems

Each student subsystem slot provides an experimental area in which a project team can implement a custom subsystem. Access to each subsystem is through an addressable bus. The number of subsystem slots is iteration-specific.

2.6.1 Interface

Subsystems communicate through an addressable bus, APB in this iteration of the SoC. Additional interface signals are provided, carrying configuration and interrupt information. Two of the configuration signals enable the subsystem clocks, while a third enables interrupt propagation out of the subsystem. The remaining wires are available for developer-defined use.

The interrupt signal is routed to the staff core and can be used by the controlling firmware as needed. Each subsystem also has access to the SoC GPIOs, which can be controlled from within the subsystem.

The complete port list of a student subsystem module is shown below. The APB subordinate port connects the subsystem to the SoC interconnect and exposes its internal registers to the CPU. The `ss_ctrl` bus carries the clock-enable and IRQ-enable bits driven by the controller block; bit 0 is the standard clock enable and bit 1 is the fast clock enable. The `irq` output is gated by `irq_en` before being forwarded to the CPU interrupt controller. The two PMOD GPIO ports each provide four bidirectional signals: `gpi` carries sampled input values into the subsystem, `gpo` carries output values driven by the subsystem, and `gpio_oe` is the per-pin output-enable mask.

```
module student_ss_example #(
    parameter APB_AW = 10,
    parameter APB_DW = 32
) (
    // Clock and reset
    input  logic                clk_in,
    input  logic                reset_int,

    // APB subordinate interface
    input  logic [APB_AW-1:0]   PADDR,
    input  logic                PENABLE,
    input  logic                PSEL,
    input  logic [APB_DW-1:0]   PWDATA,
    input  logic                PWRITE,
    input  logic [APB_DW/8-1:0] PSTRB,
    output logic [APB_DW-1:0]   PRDATA,
    output logic                PREADY,
    output logic                PSLVERR,

    // Subsystem control (from controller block)
    input  logic                irq_en_4,
    input  logic [7:0]          ss_ctrl_4,

    // Interrupt (to CPU interrupt controller)
    output logic                irq_4,

    // PMOD GPIO port 0 (4-bit bidirectional)
    input  logic [3:0]          pmod_0_gpi,
    output logic [3:0]          pmod_0_gpo,
    output logic [3:0]          pmod_0_gpio_oe,

    // PMOD GPIO port 1 (4-bit bidirectional)
    input  logic [3:0]          pmod_1_gpi,
    output logic [3:0]          pmod_1_gpo,
    output logic [3:0]          pmod_1_gpio_oe
);
```

2.6.2 Configuration

Initial control sequence:

1. Enable clock for subsystem and ICN
2. Lift ICN and Subsystem reset

3. Access subsystem with hardware memory address

If interrupts or GPIOs are required, they must be enabled through the staff section control registers.

CHAPTER

THREE

SOC SIMULATION AND VERIFICATION ENVIRONMENT

The Didactic SoC is a system integrating multiple components, including a RISC-V processor core and a debug module which are connected to SRAM memories and APB peripherals through several communication buses. The functional simulation of the Didactic SoC is the first step toward verifying that all modules operate correctly and communicate as expected. This chapter describes the complete simulation flow, from the compilation of test programs to the execution and verification of the SoC behavior.

This simulation is performed using a SystemVerilog testbench and Questa functional simulator from Siemens. The testbench first establishes communication with the Didactic SoC through the JTAG interface, which provides access to the Debug Module Interface (DMI) of the SoC by means of a dedicated shift register in the JTAG Test Access Port (TAP). The debug module is then used to halt the core, configure the SoC, initialize the memories with a software program and set the program counter to the program entry point before execution. Once the core operation is resumed, the execution of the program stimulates the communication paths from the core to memories and peripherals, verifying the correct operation of the SoC in the process. The core finally signals the termination of the program by setting a flag and writing a return status into a debug module register, which can be polled from the testbench through the JTAG interface to confirm that the program completed successfully.

The following sections cover each step of the simulation flow in detail. Section 3.1 presents the software compilation of the test programs provided as part of the Didactic SoC platform, while Section 3.2 addresses the compilation and elaboration of the SoC testbench that instantiates the SoC and peripheral modules. Last but not least, Section 3.3 walks through the simulation process implemented by the testbench.

3.1 Software compilation

The Didactic SoC platform provides several test programs to verify the SoC functionality. Namely, it provides a `blink` test program that stimulates the general purpose input/outputs pins (GPIOs) of the SoC which can be connected to LEDs to visualize the switching activity of the pins, and a `hello` test program that writes `"hello from didactic!\n"` on the UART, validating the communication with the SoC.

3.1.1 Code organization

Each program consists of a single C source file, complemented by the platform-specific startup file `crt0.S` and linker script `link.ld`, which are automatically included by the build system. The source of the test programs must be located in the `<ROOT_DIR>/sw` folder and organized according to the following convention, where `TESTCASE` is the name of a program:

```
<ROOT_DIR>/
└── sw/
```

(continues on next page)

(continued from previous page)

```
    ├── common/
    │   ├── crt0.S
    │   ├── link.ld
    │   ├── soc_ctrl.h
    │   ├── spi.h
    │   └── uart.h
    ├── <TESTCASE>/
    │   └── <TESTCASE>.c
    └── Makefile
```

3.1.2 Compilation and linking

The programs are compiled using the RISC-V GNU toolchain. The following flags are used to target the RV32IMC instruction set architecture with the ILP32 data model implemented by the Didactic SoC architecture: `-march=rv32imc -mabi=ilp32`.

Note that the runtime environment provided with the platform does not support the C standard library at the moment, and as a result dynamic memory allocation is not available by default. Support for dynamic memory allocation can however be added by providing a custom memory allocator and modifying the linker script to reserve a heap region in the data memory (DMEM).

The build process starts with the compilation of the C and assembly source files into object files:

```
riscv32-unknown-elf-gcc -marchrv32imc -mabi=ilp32 -Icommon/ -c $(TESTCASE)/
↪$(TESTCASE).c -o build/sw/$(TESTCASE).o
riscv32-unknown-elf-gcc -marchrv32imc -mabi=ilp32 -Icommon/ -c common/crt0.S -
↪o build/sw/crt0.o
```

It is followed by a linking step that produces a single ELF binary:

```
riscv32-unknown-elf-gcc -marchrv32imc -mabi=ilp32 -Tcommon/link.ld build/sw/
↪$(TESTCASE).o build/sw/crt0.o -nostartfiles -nostdlib -Wl,--gc-sections -o␣
↪build/sw/$(TESTCASE).elf
```

The linking step is controlled by the `link.ld` linker script that defines the memory layout expected by the Didactic SoC platform. The script maps the 4 kB instruction memory (IMEM), starting at address `0x01000000`, and the 4 kB data memory (DMEM), starting at address `0x01010000`. The former contains the reset vectors, program instructions and constants, placed respectively in the .vectors, .text and .rodata sections. The latter is divided between a 3 kB stack and 1 kB of statically allocated global data consisting of initialized data in the .data and .sdata sections, and uninitialized data in the .bss section.

The build process ends after converting the ELF binary into a `<TESTCASE>.hex` hex file with little-endian byte ordering and 32-bit word-aligned addresses that is compatible with SystemVerilog's `$readmemh` format using the `elf2hex` utility.

3.1.3 Execution flow

The execution of a test program follows a fixed sequence defined by the startup file `crt0.S`:

```
reset_handler → main → postMain
```

This sequence first initializes the stack and sets the `ra` return address register to the address of the `postMain` cleanup routine before transferring control to the `main` function defined in the C source file

of the test program. The `main` function then executes and eventually returns a status code in register `a0` to the `postMain` routine. The return value of the main function indicates whether the program succeeded or failed. Upon return from the `main` function, the `postMain` routine writes the return value into a dedicated debug module register at address `0x01020380` and sets its most significant bit to signal the completion of `main`. As a result, the end of the program execution can be detected by polling this register over the JTAG interface.

3.1.4 Testbench integration

The build system produces a memory initialization file at `<ROOT_DIR>/build/sw/<TESTCASE>.hex`. The path of the memory image is first passed to the testbench through parameter `TESTCASE` during the elaboration phase. The testbench then reads the image into an internal stimuli array, which is later used as the source for the firmware loading sequence:

```
parameter string HEX_LOCATION = "../build/sw/";
logic [31:0] stimuli [100000:0];
// ...
$readmemh({HEX_LOCATION,TESTCASE,".hex"}, stimuli);
```

3.2 Testbench compilation and simulation

The SoC testbench instantiates the top level of the Didactic SoC along with behavioural models of external peripherals, and defines the stimuli applied to the SoC during simulation. It is compiled and simulated using the Questa functional simulator from Siemens.

Bender is used as a hardware dependency manager to specify external dependencies and define the complete list of source files required by the project. The dependency configuration is defined in the `Bender.yml` file at the root of the project.

Before compiling or simulating the design, third-party IPs must be fetched using the following commands from the project root directory:

```
bender update
bender vendor init
```

The fetched dependencies are stored in the `<ROOT_DIR>/.bender/` folder, which is automatically populated and should not be manually modified.

The complete list of source files on which the project depends can then be generated using the following command:

```
bender script flist -t rtl -t vendor -t tracer -t didactic_obi
```

The `-t` flags select the source files associated with specific target groups: `rtl` and `vendor` include the design and third-party IP sources respectively, and `tracer` and `didactic_obi` include additional platform-specific components.

The testbench is compiled and simulated using a Makefile located in the `<ROOT_DIR>/sim/` folder. The flow is organized into three successive targets which can be built individually or in sequence: `compile`, `elaborate` and `run_sim`. The `TESTCASE` variable must be set to the name of the software project to simulate:

```
make compile
make elaborate TESTCASE=blink
make run_sim TESTCASE=blink
```

Each step produces a timestamped log file for debugging purposes in the <BUILD_DIR>/logs/compile, <BUILD_DIR>/logs/vopt and <BUILD_DIR>/logs/sim directories respectively.

3.2.1 Compilation

The compile target initializes a Questa simulation library and compiles all RTL and testbench source files into it using vlog. The list of source files is generated automatically by Bender, as described previously.

Behavioral models of external peripherals can be selectively instantiated to emulate UART and SPI devices and verify the correct operation of the corresponding interfaces. Their instantiation is controlled by the following parameter definitions in the Makefile:

- +define+USE_UART: instantiates the uart_tb_rx module, which captures UART characters transmitted by the SoC and writes them to the stdout/uart file. When disabled, the UART interface is connected in loopback configuration.
- +define+USE_SPI: instantiates the spi_slave module, which provides helper tasks to receive and transmit data to the SoC once selected.

3.2.2 Elaboration

The elaborate target elaborates and optimizes the compiled design using vopt, setting tb_didactic as the top-level module. Two parameters are passed to the testbench at this stage:

- -gTESTCASE=<TESTCASE>: specifies the name of the software project to simulate, used by the testbench to locate the corresponding hex file in <ROOT_DIR>/build/sw.
- -gDM_SANITY_TESTCASES=<SANITY_CHECK>: perform debug module sanity checks during simulation when enabled and load the test program in SoC memories through the JTAG interface when disabled. Defaults to 0 (disabled).

3.2.3 Simulation

The run_sim target launches the simulation on the elaborated snapshot using vsim. It first creates the stdout/uart file to capture UART output from the SoC, then starts the simulator. An initial run 0ms command is issued to trigger design initialization without advancing simulation time, after which a run -all command should be manually issued to drive the simulation autonomously until the program signals its termination through the debug module register.

3.3 Simulation flow

The simulation flow is driven by a sequential process in the testbench which communicates with the Didactic SoC through the JTAG interface. This process follows a pre-defined sequence of operations that can be divided into five phases:

1. reset phase: the SoC is brought out of reset ;
2. JTAG initialization phase: the debug session is established and the core is halted ;
3. Firmware loading phase: the program is written into the instruction memory ;
4. Execution phase: the program counter is set to the entry point and the core is resumed ;

5. Monitoring phase: the testbench polls the SoC for program termination and retrieves the exit status.

3.3.1 Reset sequence

The simulation begins with a reset sequence that brings the SoC into a known state before any JTAG transaction is initiated. The reset signal is first asserted for 3 ms, then released for an additional 3 ms to emulate an external negative reset and the subsequent settling time for the SoC. Only after this period does the testbench begin communicating with the SoC through the JTAG interface.

3.3.2 JTAG interface validation

Before opening a debug session, the testbench performs a series of preliminary checks to validate the JTAG interface. The TAP controller is first brought to a known state using `jtag_reset`, which asserts the `TRST` signal, followed by `jtag_softreset`, which drives the TAP state machine to the Test-Logic-Reset state by asserting TMS for five clock cycles. The bypass register is then tested using `jtag_bypass_test` to verify that the shift path through the TAP is functional. Finally, the IDCODE register is read using `jtag_get_idcode` and compared against the expected value `0x1C0FFEE1` to confirm that the correct device is connected.

All communication with the debug module is carried out through the Debug Module Interface (DMI), which is accessed via a dedicated shift register (DMIACCESS) in the JTAG TAP controller. Each DMI transaction consists of a register address, a data field and an operation code, which are shifted into the TAP controller's internal shift register on the JTAG clock domain. The transaction is then posted through a clock domain crossing (CDC) circuit to the debug module, which operates on the system clock.

3.3.3 Debug module initialization

Once the JTAG interface is validated, the testbench initializes the debug session. The CONFREG instruction is first selected via the `test_mode_if` interface to configure the SoC. The DMI data register is then selected for subsequent transactions by calling `init_dmi_access`, which loads the `DMIACCESS` opcode into the JTAG instruction register. The debug module is activated by asserting `dmactive` in the `DMControl` register via `set_dmactive`, after which the target core is selected using `set_hartsel` and halted using `halt_harts`, which asserts `haltreq` in `DMControl`. The `DMStatus` register is polled until the `allhalted` flag is set, confirming that the core has stopped execution. Finally, the boot address is written into the `dpc` register using `write_reg_abstract_cmd`, setting the program counter to the firmware entry point before resuming the core.

3.3.4 Debug module sanity check

If the `DM_SANITY_TESTCASES` parameter is set to a non-zero value at elaboration time, the testbench runs a series of debug module functional tests instead of loading the firmware. These tests stimulate the core debug module functionalities including core discovery, halt and resume, register access via abstract commands, program buffer execution, single stepping and memory access through the OBI system bus interface. This optional step is intended to verify that the debug module itself operates correctly before relying on it to load and execute a program.

3.3.5 Firmware loading

Once the debug session is established and the core is halted, the firmware is loaded into the instruction memory (IMEM) through the system bus access interface if the `DM_SANITY_TESTCASES` parameter is set to zero at elaboration time (default value). The `load_L2` task iterates over the contents of the hex file generated during the software compilation step and writes each 32-bit word to the target address in IMEM, starting from the base address `0x01000000` and incrementing by four bytes after each write.

3.3.6 Core release and execution

Once the firmware is loaded and the program counter is set to the entry point, control is handed back to the core by calling `resume_harts`, which asserts `resumereq` in `DMControl` and polls `DMStatus` until the `allresumeack` flag is set, confirming that the core has resumed execution. The core then begins executing the firmware from the address stored in the `dpc` register.

3.3.7 Execution monitoring and termination

Once the core is running, the testbench monitors the execution by periodically polling the debug module register at address `0x01020380` every 100 µs. The termination of the program is detected when bit 31 of the register is set by the `postMain` routine. The exit status is then retrieved from bits 30 to 0 of the same register and reported by the testbench, indicating whether the program completed successfully or encountered an error. The simulation is subsequently stopped.

CHAPTER

FOUR

STUDENT SUBSYSTEM FRONT-END INTEGRATION AND VERIFICATION

All student subsystems share a common interface. This allows for a reusable common testbench design. The basic verification setup can be copied from the provided example and only minor adjustments have to be made in order to adapt the base testbench to the implemented subsystem. The base testbench allows for quick development of constrained random tests for any student subsystem, as the simulation environment, bus interface drivers, and basic test setup are already present and can be reused.

The base testbench in implemented in the Didactic-SoC repository. It contains the reusable base testbench components, as well as some examples on how to implement the DUT specific components and DUT specific testcases.

In the following, it is assumed that the repository root is named <REPO_ROOT> and the root of the example test bench is named <SSTBEX_ROOT> which is <REPO_ROOT>/verification/student_ss. In the provided example the RTL code being tested is the <REPO_ROOT>/src/rtl/student_ss_example.sv.

4.1 Student subsystem wrapper

The student testbench relies on a common wrapper module located in <SSRBEX_ROOT>/common/tb_top/top.sv. The DUT is instatiated within this module, and the I/O signal are connected to the top module. This ensures the correct connection of the I/O to the testbench signals.

4.1.1 I/O Description

The student subsystems feature an Advanced Peripheral Bus interface (APB), control signals and 2x4 configurable IOs as pmod_0 and pmod_1 respectively. The APB Interface has paratemizable address and data widths as described in Table 4.1.

Table 4.1: APB Parameters

Parameter Name	Parameter
APB_AW	APB interface address width
APB_DW	APB interface data width

Table 4.2: APB Signals

Signal name	Direction	Width
PADDR	input	[APB_AW-1:0]
PENABLE	input	1
PSEL	input	1
PWDATA	input	[APB_DW-1:0]
PWRITE	input	1
PSTRB	input	[APB_DW/8-1:0]
PRDATA	output	[APB_DW-1:0]
PREADY	output	1
PSLVERR	output	1

The student subsystem also features eight GPIO pins, clock, reset, interrupt, and control signals as outlined in Table 4.3. The direction of the port is taken from the perspective of the student subsystem.

Table 4.3: Student subsystem general signals

Signal name	Direction	Width
clk_in	Input	1
reset_int	Input	1
irq_4	Output	1
irq_en_4	Input	1
ss_ctrl_4	Input	[7:0]
pmod_0_gpi	Input	[3:0]
pmod_0_gpo	Output	[3:0]
pmod_0_gpio_oe	Output	[3:0]
pmod_1_gpi	Input	[3:0]
pmod_1_gpo	Output	[3:0]
pmod_1_gpio_oe	Output	[3:0]

Student subsytem example

The example testbench implements testcases for a very simple DUT. The example is located in <SSTBEX_ROOT>/src/rtl/student_ss_example.sv.

The DUT implements a very simple APB slave with some added GPIO and control signal functionality. Internally the DUT features three physical registers: rw_reg, gpio_w_reg and ss_ctrl_reg. These registers are mapped on memory adresses 0x00, 0x08 and 0x0C``respectively, as described in :numref:`table_toy_ss_registsers`. The state of the GPIOs is mapped on address ``0x04 and can be read by accessing the virtual register gpio_r_reg.

Table 4.4: Subsytem example register map and fields

Address	Register	Description	Bits	Field	Name	Rst.	Access
0x00	rw_reg	32-bit register for reading and writing	[7:0]	field0	field0	0	Read-/Write
			[15:8]	field1	field1	0	Read-/Write
			[31:16]	field2	Unused	0	Read-/Write
0x04	gpio_r_reg	32-bit register for reading of GPIOs	[3:0]	gpio0_fie	GPIO_0	0	Read-/Write
			[7:4]	gpio1_fie	GPIO_1	0	Read-/Write
			[31:8]	unused	Unused	0	Read-/Write
0x08	gpio_w_reg	32-bit register for writing	[3:0]	gpio0_fie	GPIO_0	0	Read-/Write
			[7:4]	gpio1_fie	GPIO_1	0	Read-/Write
			[31:8]	unused	Unused	0	Read-/Write
0x0C	ss_ctrl_reg		[7:0]	ss_ctrl	SS_CTR	0	Read-/Write
			[31:8]	unused	Unused	0	Read-/Write

The content of all registers can be accessed thought the APB interface. `rw_reg` can only be accessed thoug the APB interface. `gpio_r_reg` contains the input data from GPIO 0/1 in the respecive field, if they are configured to be inputs. `gpio_w_reg` is written to the GPRIO 0/1 pins, if they are configured as outputs. The lower eight bits of the `ss_ctrl_reg` will always contain the input from the `ss_control` signal.

Subsystem simulation using cocotb

The testbench is written in Python using a tool chain using the following three major tools. The RTL design is simulated using Icarus Verilog. Cocotb is used as a co-simulation tool allowing Python to interact with the simulation. Lastly pyuvm implements the Universal Verification Methodology (UVM) framework in Python. The tool stack can be seen in Fig. 4.1

The installation of the tool stack is documented in `<SSTBEX_ROOT>/readme.md`.

Functional verification primer

The testbench implements the basics to functionally verify the DUT using constrained random verification. The verification is split up into three independent parts:

1. Stimuli generation
2. Response checking
3. Coverage measurement

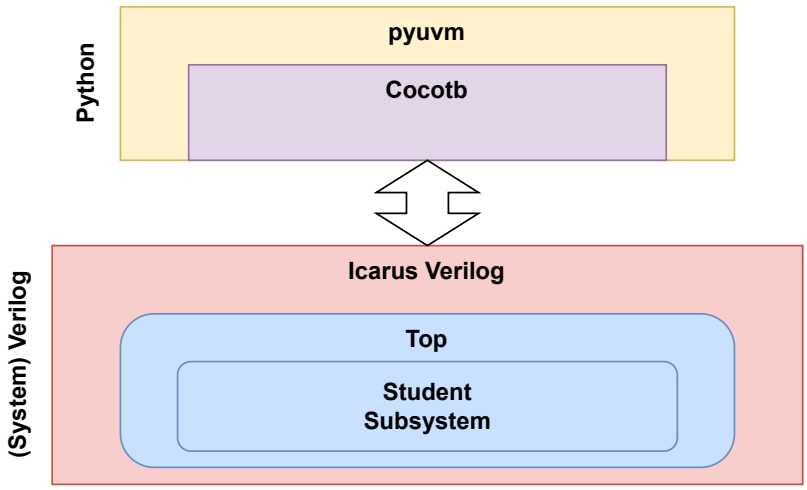

Fig. 4.1: Tool stack used for simulation.

4.1.2 Stimuli generation

Constrained random (stimuli generation) allows for the automatic generation a large number of tests. Instead of writing directed tests, i.e. a set of input stimuli, which exercises a sequence of states in the DUT to test a specific feature, input stimuli are randomly generated. Since truly random stimuli are unlikely to produce interesting tests the generation of stimuli is constrained.

4.1.3 Response checking

Typically the response of the DUT is checked using reference models and assertions. The example testbench has not reference model, but provides an example assertion for the interrupt behavior of the DUT in `<SSTBEX_ROOT>/common/cl_student_tb_assertions.py`.

4.1.4 Coverage measurement

Coverage is the measurement of the verification effort. It measures if all desired states of the DUT have been observed. A good testbench is only as good as the employed coverage model. The testbench also uses the pyvsc library for coverage, in addition to randomization.

4.1.5 Transaction level modeling

UVM provides additional abstraction levels for the stimuli generation. Pin level inputs are abstracted using transaction level modeling (TLM). TLM abstracts the pin level signal into individual transactions. A transaction is the minimal description of a bus event. Instead of individually driving the pins of the APB interface transaction can be expressed on a higher abstraction level as a single APB transaction. A APB Transaction, for example, consists of the following fields:

Name	Description
Operation Type	Read/Write Operation
Address	Integer in range `[0,APB_AW*8]`
Data	Integer in range `[0,APB_DW*8]`
Write Strobe	Integer in range `[0,APB_DW]`
Slave Error	Transfer failure

To randomize a transaction, the individual fields of the transaction are randomized, then a driver translates the transaction to the pin level inputs. The testbench typically interacts with the DUT interface through Universal Verification Components (UVC).

Testbench explanation

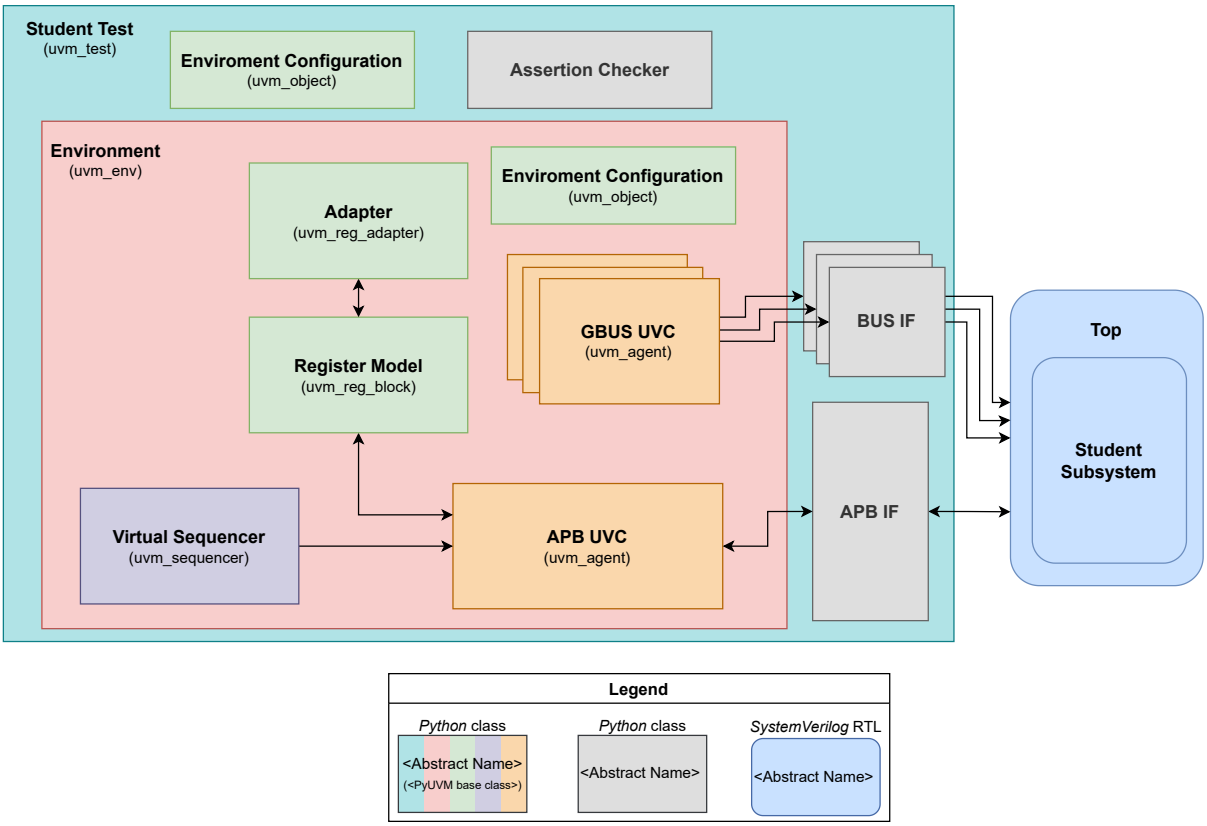

Fig. 4.2: Student subsystem testbench overview

The majority of the student testbench is reusable between different subsystems. In the following sections all parts of the testbench which needs to be configured to adapt the testbench to a new DUT are explained. These parts are highlighted in Fig. 4.2.

UVCs

The testbench has two types of UVCs: the APB UVC, and a general bus (GBUS) uvc. They are located in <SSTBEX_ROOT>/common/uvc.

4.1.6 GBUS UVC

The GBUS UVC offers an easy interface to the bus interfaces of the student subsystem. The student subsystem testbench has GBUS UVCs configured for all bus signals. The I/O signals of the student subsystem are grouped into the following busses:

Bus	Signals
PMOD 0 GPI	pmod_0_gpo
PMOD 0 GPO	pmod_0_gpi
PMOD 1 GPI	pmod_1_gpi
PMOD 1 GPO	pmod_1_gpo
SS CTRL	ss_ctrl_4
IRQ EN	irq_en_4
IRQ	irq_4

The respective agents are instantiated in the environment in <SSTBEX_ROOT>/common/cl_student_tb_env.py.

The pins of each bus can be driven using the sequences provided in <SSTBEX_ROOT>/common/uvc/gbus/cl_gbus_seq_lib.py. Additionally the base sequence in <SSTBEX_ROOT>/common/cl_student_base_sequence.py offers two functions sample_gpo_pins() and drive_gpi_pins() to drive and sample the GPIO pins respectively.

4.1.7 APB UVC

The APB UVC drives and samples the APB interface of the subsystem. It is connected to the register model though the register adapter. The sequences on the UVC are started automatically whenever the register model receives a read or write.

4.1.8 Register Model

The testbench uses a register model for the internal registers of the subsystem. Registers are modeled with the uvm_reg and uvm_reg_block classes. Each register consists of a number of fields. The registers are group together in a register block, which also contains the register map and adapter. Each register field needs to be configured, with the following parameters[1]:

1. Field size
2. Least significant bit position
3. Access type
4. Volatility
5. Reset value

There are more optional parameters, see [IEE20] section 18.5 for more optional information. The configuration for the rw_reg is shown in Listing 4.1.

[1] The parameters are shown in the order of arguments in the .configure() call.

Listing 4.1: UVM Register definition for `rw_reg`

```
class reg0(uvm_reg):
    """Class : The UVM register class for reg0"""

    def __init__(self, name="reg0",reg_width=32):
        super().__init__(name, reg_width)

        self.field0 = None
        self.field1 = None
        self.field2 = None

    # Build : The build function of reg0.
    def build(self):

        # Constructing all fields in the current register.
        self.field0 = uvm_reg_field("field0")
        self.field1 = uvm_reg_field("field1")
        self.field2 = uvm_reg_field("field2")

        # Configuring all fields of the current register.
        self.field0.configure(self, 8, 0, "RW", True, 0)
        self.field1.configure(self, 8, 8, "RW", True, 0)
        self.field2.configure(self, 16, 16, "RW", False, 0)
```

The inidividual registers are then grouped together in a `uvm_reg_block`. See `<SSTBEX_ROOT>/common/register_model/student_ss_regs_reg_block.py` for reference.

In the testbench environment, the register model (register block) is connected to the APB UVCs register adapter. The APB UVC then mirrors the read and write accesses to the register model on the DUT.

4.1.9 Writing and reading from registers

To write and read registers, the virtual sequencer in `<SSTBEX_ROOT>/common/cl_student_tb_vsequencer.py` has two functions `reg_write()` and `reg_read()`. `<SSTBEX_ROOT>/common/register_model/cl_reg_dynamic_seq.py` shows how a random read/write sequence can be implemented.[2]

Adapting the testbench for a new design

To adapt the testbench to a new design, most of the `<SSTBEX_ROOT>/common` folder can be reused.

1. Updating the register model
2. Adapting the virtual sequences
3. Updating the RTL (see `<SSTBEX_ROOT>/readme.md` for more information)

[2] Note the APB sequence item is only instantiated for easy data generation. The APB UVC is connected though the register adapter and automatically sends APB transaction to the DUT based on the write/read requests to the register model.

4.1.10 Updating the register model

The register model is specific to the example DUT and this needs to be changed to reflect the DUT.

The following files need to be updated:

- The register model (<SSTBEX_ROOT>/common/register_model/student_ss_regs_regs.py)
- The register block (<SSTBEX_ROOT>/common/register_model/student_ss_regs_block.py)
- The dynamic register sequence (<SSTBEX_ROOT>/common/register_model/cl_reg_dynamic_seq.py)
- Subsystem control register sequence (<SSTBEX_ROOT>/common/register_model/cl_reg_ss_ctrl_seq.py)

4.1.11 Adapting the virtual sequences

The virtual sequences in <SSTBEX_ROOT>/common/sequences are kept general such that they can apply to most designs, still they should be checked for applicability to a new design before used, and if necessary adapted.

4.1.12 Writing new testcases

Once the base testbench is adapted to the new subsystem new testcases can be implemented. Under <SSTBEX_ROOT>/tests a few different example testcases are shown.

Any testcase consists of a test-file and virtual sequence (vseq) file. The virtual sequences are placed in <SSTBEX_ROOT>/common/sequences and defines the behavior of the testcase by calling other sequences and starting them.

The test-file should start the sequence on the sequencer and can define assertions for checking results if desired.

CHAPTER

FIVE

SOC INTERFACE AND HW/SW CO-DESIGN

The Didactic SoC developed within the Edu4Chip project is designed to provide you with hands-on experience in SoC integration, combining hardware design with embedded software development. Didactic integrates processors, memories, specific-purpose subsystems, and peripherals into a single chip, requiring efficient communication mechanisms and clear separation between hardware and software layers. The goal of this section is to explain how the firmware is developed in the context of the Didactic SoC and how hardware/software interact with each other.

Developing a firmware for the Didactic SoC involves a structured approach that builds upon the interaction between software and memory-mapped hardware components. The process begins with proper initialization of the platform and the targeted subsystem, followed by the use of abstraction layers to simplify hardware access, and finally the implementation of coordinated hardware/software interaction to perform the desired functionality. This workflow ensures that subsystems can be configured, controlled, and monitored efficiently through standardized interfaces. The following sections detail this process. Section 5.1 describes how the platform and subsystems are initialized. Section 5.2 introduces the Hardware Abstraction Layer (HAL) used to encapsulate low-level register access. Section 5.3 explains how hardware and software are co-designed to enable seamless subsystem integration and operation.

5.1 Platform and Subsystem Initialization

The platform initialization typically is responsible for setting up the essential hardware infrastructure before executing the main application.This process typically includes configuring the system clocks, initializing memory regions, and initializing peripherals and subsystems through a memory-mapped I/O scheme. The processor starts execution from a reset vector stored in ROM, which redirects execution to a reset handler. In general, the reset handler performs several key steps, including initializing the stack pointer, setting up the runtime environment, configuring interrupt handling, and enabling communication buses. In addition to these steps, the correct placement of code and data in memory is ensured by the linker script (`link.ld`), which defines how different program sections (such as `.text` and `.data`) are mapped into the available instruction and data memory regions. The linker script organizes the program into standard sections, where the `.text` section contains executable code, the `.data` section stores initialized global variables, and the `.bss` section holds uninitialized variables that are automatically set to zero at runtime, and the `.stack` section defines the memory region used for function calls and local variables during program execution. It also specifies the location of the stack and important symbols used during program execution, ensuring that the processor accesses instructions and data from the intended memory areas.

The Didactic SoC adopts a simplified configuration only with the essential steps required to start program execution aimed at educational clarity and ease of understanding. As can be seen in `ctr0.S`, the reset vector directly jumps to a `reset_handler`, which initializes the stack pointer and transfers control to the `main` function. This implementation does not include interrupt support, and the vector table contains only basic entries, a default handler and the reset vector. After the main program completes, execution

returns to a dedicated `postMain` routine. It export the result of the program to the outside world through a memory-mapped register accessible via JTAG.

Each subsystem, developed by different partners, is integrated as a memory-mapped IP block. For each subsystem, a predefined memory map specifies the base addresses and register offsets used for control and data exchange. Configuration steps depends on the specific subsytems and typically include setting control registers, enabling or disabling features, and initializing internal states. This is done by writing to registers exposed through the APB interface, making the subsystem behave like a standard peripheral.

The global memory organization of the SoC is summarized in the Table 5.1. It defines the address ranges assigned to instruction memory, data memory, control registers, and each subsystem slot. In particular, student subsystems are mapped within the interconnect region starting at address `0x0105_0000`, with each subsystem occupying a dedicated address range. By referring to this memory map, firmware developers can determine the correct base addresses and offsets required to access and configure each subsystem during initialization.

Table 5.1: Memory map

Memory map target	Start	End	Actual size
Instruction memory	0x0100_0000	0x0100_FFFF	16 KiB
Data memory	0x0101_0000	0x0101_FFFF	16 KiB
Debug module	0x0102_0000	0x0102_FFFF	0x900
Staff peripherals	0x0103_0000	0x0103_FFFF	0x300
Control registers	0x0104_0000	0x0104_FFFF	0x184
Interconnect	0x0105_0000	0x0105_FFFF	Denoted by number of ss
Student 0	0x0105_0000	0x0105_0FFF	
Student 1	0x0105_1000	0x0105_1FFF	
Student 2	0x0105_2000	0x0105_2FFF	Template is empty
Student 3	0x0105_3000	0x0105_3FFF	Template is empty

5.2 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) provides a structured interface to configure and interact with hardware components within the SoC. Its primary purpose is to isolate low-level register manipulations from application-level software, enabling developers to interact with peripherals through well-defined and reusable APIs. The HAL acts as a wrapper around low-level drivers, enabling portability and maintainability across platform and subsystems.

In the Didactic SoC, the HAL is designed using a modular approach, where each peripheral or subsystem is encapsulated within a dedicated header file. Each module typically provides Register definitions (base addresses and offsets), Macros for memory-mapped access, a set of control functions that directly manipulate memory-mapped control registers.

The common HAL modules include:

- Control HAL(`soc_ctrl.h`): global SoC and subsystem control
- GPIO HAL (`gpio.h`): Provides functions to configure pins as input/output and read/write digital values.
- UART HAL (`uart.h`): Enables serial communication through functions for initialization, transmission, and reception.
- SPI HAL (`spi.h`): Supports communication with external devices using the SPI protocol.

5.2.1 Control HAL (soc_ctrl.h)

The Control HAL provides low-level access to global SoC configuration registers responsible for managing subsystem reset, clock control, and external routing. It is based on a set of memory-mapped registers defined relative to a control base address (CTRL_BASE), where each subsystem is associated with a dedicated control register. It provides several abstraction functions. Functions such as ss_init() and ss_init_high_speed() enable a target subsystem by configuring its reset state and activating its clock, while ss_reset() disables the subsystem and clears its configuration. In addition, the pmod_target() function allows routing of external I/O (PMOD) signals to a selected subsystem.

This HAL encapsulates bit-level manipulation of control registers, allowing software to initialize and manage subsystems through simple function calls rather than direct register access. It plays a central role in platform initialization and subsystem activation within the Didactic SoC.

5.2.2 GPIO HAL (gpio.h)

The GPIO HAL provides an interface for configuring and controlling general-purpose input/output pins. It defines memory-mapped registers for reading input values (PAD_IN) and writing output values (PAD_OUT), as well as a set of configuration registers used to control the direction and behavior of each GPIO pin. Functions such as gpio_init_out() and gpio_init_in() configure individual pins as outputs or inputs by updating the corresponding pad configuration registers.

Once configured, the GPIO pins can be accessed through gpio_write() to drive output values and gpio_read() to sample input signals. This HAL enables straightforward interaction with external hardware components while abstracting the underlying register-level operations.

5.2.3 UART HAL (uart.h)

The UART HAL provides a minimal interface for serial communication through a set of memory-mapped registers controlling transmission, reception, and configuration. The uart_init() function configures the UART by setting the baud rate divisor, enabling FIFO buffers, and initializing control registers for standard communication. It also configures the corresponding I/O pad for reception.

Data transmission is handled by the uart_print() function, which sends characters sequentially by writing to the transmit register. Due to the absence of interrupt-based handling in this implementation, a simple delay loop is used to ensure correct timing between transmissions. Additionally, a basic uart_loopback_test() function is provided to verify functionality by transmitting and reading back a character.

5.2.4 SPI HAL (spi.h)

The SPI HAL provides functionality for communicating with external devices using the Serial Peripheral Interface protocol. It defines a set of memory-mapped registers for controlling SPI transactions, including command configuration, data length, address, and FIFO buffers for data transfer. The spi_init() function configures the pad settings for SPI signals, preparing the interface for communication.

The HAL supports both read and write operations through spi_read() and spi_write() functions. These functions configure the SPI transaction parameters, set the appropriate command type, and control the direction of the data pins by updating pad configuration registers. Data is transferred through transmit and receive FIFOs, while control and status registers manage the execution of the transaction.

5.3 Hardware/Software co-design through practical examples

The previous subsection introduced the common HAL modules used for system control and standard peripherals. Building on those, this subsection focuses on how individual subsystems are integrated and operated within the SoC.

In the Didactic SoC, hardware/software co-design is realized through a clear and complementary separation of responsibilities. The hardware defines the available interfaces such as memory-mapped registers and communication protocols while the software is responsible for configuring and controlling these interfaces.

As mentioned before, each partner university develops a dedicated subsystem that is connected to the main system through a standard APB bus interface and exposed to the CPU via memory-mapped registers. The interaction between hardware and software follows a simple and consistent model: the hardware implements control and data registers, the software accesses these registers through read/write operations, and the hardware executes the requested functionality. Completion of operations is typically handled through polling, where the software repeatedly checks a status register until the computation is finished. Some of the systems are not open-source. They are described here as an example of what could be done with the Didactic-SoC platform.

5.3.1 DTU subsystem

The DTU subsystem integrates Leros, a lightweight 32-bit processor core described in Scala and generated for inclusion in the Didactic SoC as an APB peripheral. Leros operates as a secondary programmable processor, independent of the main Ibex core, with its own instruction memory and data memory.

On the hardware side, the subsystem exposes three categories of memory-mapped components: instruction memory (`IMEM`), system control registers (`SYS_CTRL`), and cross-core registers (`REGS`). The instruction memory holds the program executed by the Leros processor. The system control registers manage processor reset, boot source selection (ROM or RAM), and UART loopback mode. The cross-core registers provide a bidirectional data exchange mechanism between the Leros core and the Ibex core.

The subsystem is first initialized through the SoC control HAL by enabling the subsystem clock and releasing the reset signals. The Ibex firmware then writes a program directly into `IMEM` via the APB interface and configures the boot source through `SYS_CTRL`. Releasing the Leros processor from reset starts its execution.

During operation, the Ibex core exchanges data with Leros through the cross-core registers and monitors execution progress by polling the registers. Once the Leros program signals completion, the Ibex core reads the result registers and verifies correctness. This setup demonstrates a flexible hardware/software co-design approach, where a secondary processor is integrated into the SoC and interacts with the main CPU through standardized memory-mapped interfaces.

5.3.2 KTH subsystem

The KTH subsystem implements a 16-point FFT accelerator developed using the SiLago design workflow, which targets a Coarse-Grained Reconfigurable Architecture (CGRA).

On the hardware side, the subsystem exposes three categories of memory-mapped registers:

- Instruction registers, used to load the CGRA configuration sequence
- Control registers, used to start execution and configure the accelerator
- Data registers, used to supply input samples and retrieve output results

Dedicated call and return registers are used to trigger execution and signal its completion.

The subsystem is first initialized through the SoC control HAL by enabling the subsystem clock and releasing the reset signals. The control register is then written to configure the number of active computation cells in the Dynamically Reconfigurable Resource Array (DRRA). The firmware subsequently writes the CGRA instruction sequence into the instruction registers and loads the input FFT samples into the data registers.

Execution is triggered by writing to the call register. The Ibex core polls the return register until the accelerator signals completion, after which the output data is read from the data registers and validated against expected values. The validation result is transmitted to the host over UART. This sequence demonstrates a complete and structured hardware/software interaction flow, including instruction loading, data transfer, execution control, result retrieval, and verification.

5.3.3 TUM subsystem

Students at TUM can select one out of three projects for implementation: A Phase Locked Loop Design in the analog domain, and an AI accelerator or a security module in the digital domain. In all cases the students have all degrees of freedom to design their subblock as long as they meet few specifications. All projects are designed to be handled by groups of five students but can be scaled to slightly larger or smaller groups.

Phase Locked Loop (PLL)

The building blocks for the PLL consist of a phase detector, a low-pass filter, a voltage-controlled oscillator (VCO) and a feedback divider. In addition, modeling of the analog block must be done by the students. Specifications for the phase locked loop are provided in form of required input and output clock frequency, supply voltage, maximum power consumption, and maximum settling time. On top, jitter should be minimized.

The Analog block is mostly independent from the rest of the design. To enable measurement and evaluation of the PLL block dedicated analog pins are included in the I/O ring, so that students can directly connect to the environment. In future the analog design project might be extended so that students can also interact with the digital domain, e.g., to configure the PLL.

AI Accelerator

For the AI accelerator, the students will implement a matrix multiplier as a loosely coupled accelerator for the main core in the staff area of the Didactic SoC. Students will have to implement building blocks like Multiply accumulate units, control logic for the array, and interfacing towards the micro controller (MC). Students will have to test the submodule and to implement corresponding embedded software on the micro controller.

The main connection of the module is to the micro controller via the APB bus interface. In addition, students can connect to a PMOD interface to output triggers for later measurement or debugging. The main non-functional specification for the AI Accelerator will be given in terms of area constraint and minimum frequency requirements.

Security Module

For the security module, students will implement a memory encryption unit as a loosely coupled accelerator for the microcontroller (MC) in the staff area. It will consist of interfaces towards the APB bus and towards the off-chip environment, a symmetric cipher, a Physical Unclonable Function (PUF) with error correction encoder and decoder and other post-processing units, and an overall control of the system. The connection towards the MC will be via the APB bus; connection to the outside world shall be implemented using a PMOD connection as either UART or SPI depending on the students' preferences.

Virtually, the PMOD interface is connected externally to a non-volatile memory (NVM); for the lab, this NVM will be emulated by a PC.

Students will have the task and the degree of freedom to decide the concrete implementation of the interfaces as well as of the other components mentioned. They will have to test their design in simulation and on FPGA and will have to write corresponding code for the MC on the chip as well as for the PC emulating the memory. The main non-functional specifications for this module will be the maximum area as well as the minimum frequency, just as for the AI accelerator.

5.3.4 IMT subsystem

The IMT subsystem implements an Ascon-AEAD128 cryptographic accelerator compliant with the NIST SP 800-232 standard. A cryptographic permutation function coupled with a finite state machine performs authenticated encryption and decryption, processing the payload one 128-bit block at a time through a handshake protocol with the main processor.

On the hardware side, the subsystem exposes three categories of memory-mapped registers:

- Configuration registers, used to supply the associated data and payload sizes, the 128-bit key, and the 128-bit nonce
- Control and acknowledge registers, used to start or stop an operation and to acknowledge block transfers during streaming
- Status and data registers, exposing busy, done, data-ready, and tag-valid flags, as well as the 128-bit data input, output, and authentication tag banks

The subsystem is initialized through the SoC control HAL by enabling the subsystem clock and releasing the reset signals. A stop command is then written to the control register and the firmware polls the status register until the busy flag clears, ensuring the accelerator is in a known idle state before use. The 128-bit key and nonce are written into the key and nonce registers.

An encryption or decryption operation is started by writing the associated data and payload sizes to the configuration register and the operation code to the control register. The firmware streams associated data blocks first, then plaintext or ciphertext blocks, one 128-bit block at a time. For each input block, the firmware waits for the data-input-ready flag, writes the block to the data input register, and sets the input-valid acknowledge bit to signal that the block is ready. For each output block, it additionally waits for the data-output-valid flag, reads the block from the data output register, and sets the output-ready acknowledge bit. Once the done flag is set, the authentication tag is read from the tag register and the tag-valid flag is checked. This subsystem illustrates how a streaming cryptographic accelerator with a block-level ready-valid handshake can be integrated as an APB peripheral.

CHAPTER

SIX

FPGA VERIFICATION

This chapter explains why FPGA-based validation is a critical step in an SoC development flow, what is covered by FPGA testing, and how to test the Didactic-SoC on a Xilinx FPGA.

6.1 Why FPGA-Based Validation

FPGA-based verification of the Didactic-SoC ensures that the system can boot, execute software, and interact with real peripherals on a physical platform before committing to silicon. It exposes the design to realistic timing, clocking, and I/O conditions. While RTL and gate-level simulations are essential, they operate under idealized conditions and limited execution time. They cannot fully capture long-running system behavior or interactions with real hardware interfaces.

ASIC development is inherently high risk: tape-out and fabrication are expensive and time-consuming, post-silicon debugging offers limited observability and slow iteration, and design errors frequently require a costly re-spin. FPGA prototyping mitigates these risks by enabling early system-level validation. Designs can run at tens of MHz, allowing execution of long-duration tests that are impractical in simulation.

Furthermore, FPGA testing exposes the design to non-ideal effects such as clock quality, reset sequencing, and I/O timing, which are typically abstracted in simulation. It also enables interaction with real peripherals and lab equipment, allowing validation of system-level assumptions and debugging workflows.

For the Didactic-SoC, FPGA verification demonstrates that the system can:

- Reset reliably.
- Boot via JTAG.
- Execute bare-metal RISC-V software.
- Support debugging over JTAG.
- Communicate over basic peripherals (UART, GPIO, SPI).

In other words, FPGA prototyping turns the Didactic-SoC into a functional development platform prior to ASIC realization, which and can also be used as a target platform in an educational setting.

6.2 Implementing the Didactic-SoC on FPGA

This section provides a complete, practical workflow for building, programming, and debugging the Didactic-SoC on Xilinx FPGA platforms, from bitstream generation to software execution via JTAG.

6.2.1 Bitstream Generation

The project uses *make* automation for the FPGA flow. The default setup assumes a *bash* shell environment, so adjustments are required for Windows (e.g., using WSL or MSYS2, or adapting the Makefile for PowerShell) and possibly on macOS depending on the shell configuration.

The top-level *Makefile* provides a target called *fpga*. To use it, specify the `PROJECT` variable to select the target board:

- `z1` – PYNQ-Z1 board. Xilinx part: xc7z020clg400-1. Link to product page.
- `basys3` – Basys 3 board. Xilinx part: xc7a35tcpg236-1. Link to product page.
- `basys3_vjtag` - Basys 3 AMD Artix 7 Trainer board. Link to product page.

The `basys3_vjtag` target is intended for the Basys3 board and uses the onboard JTAG interface (normally used for bitstream programming). This is useful when no external JTAG adapter is available.

The other targets were tested with an external FT2232H MiniModule as a JTAG adapter. Other adapters can also be used, provided that the physical connections to the FPGA PMOD header and the OpenOCD configuration are adapted accordingly.

Before running the FPGA build, ensure that Xilinx Vivado is installed and its environment is available in your shell (i.e., the `vivado` command is in your `PATH`).

After running:

```
make fpga PROJECT=<target>
```

a bitstream is generated, which must then be programmed onto the FPGA.

6.2.2 Building Software

The CPU software for the FPGA is located in `fpga/sw`. This directory contains two simple examples: one for testing GPIOs and another for testing UART. To build the software, the `riscv32-unknown-elf` toolchain must be installed.

Navigate to `fpga/sw` and run:

```
make test TESTCASE=<target>
```

Currently, the available `TESTCASE` options are:

- *blinky* - Blinks all LEDs every 100k clock cycles.
- *hello* - Repeatedly transmits `"hello from didactic!"` over UART. Make sure to connect a UART-to-USB bridge to the FPGA pins specified in the `.xdc` file, and use the configured baud rate. By default, with an 8 MHz system clock, the baud rate is 9600.

Additional test cases can be added by creating a new directory with a corresponding *.c* file.

6.2.3 Hardware Setup (JTAG)

After building the software, the physical connections must be made between the FPGA and the JTAG adapter. More information on connecting the devkits can be found in `fpga/README.md` and `doc/jtag_setup.md`. You can also refer to the .xdc files in `fpga/constraints/` for pin assignments on the FPGA side. This step can be skipped when using the *basys3_vjtag* target.

6.2.4 Loading via JTAG

The CPU software is loaded using a JTAG debugger. This project uses the *riscv-dbg* debugger (https://github.com/pulp-platform/riscv-dbg), which requires:

- OpenOCD (installed separately) - use mainline OpenOCD, tagged release 0.12.0
- *riscv32-unknown-elf-gdb* (included in the toolchain)

After connecting the JTAG adapter and installing OpenOCD, start the OpenOCD server using the configuration file from `fpga/utils`:

```
openocd -f <adapter_config>
```

If you built the bitstream with the target `basys3_vjtag`, use the configuration file `fpga/utils/openocd-didactic-vjtag.cfg`. For the other targets (if using FT2232H MiniModule JTAG adapter) use `fpga/utils/openocd-didactic.cfg`. Otherwise you can also create your own adapter configuration file.

Then, from the `fpga/` directory, load the program by running:

```
make load_elf TEST=<target>
```

The `TEST` variable should match the `TESTCASE` used during the build step (or another software project that was successfully built).

Once loaded, execution can be started via GDB or through the OpenOCD telnet interface.

6.2.5 Debug via JTAG

The core can be debugged via JTAG using GDB. Typical operations include setting breakpoints, halting execution, and reading/writing registers and memory.

In the following, we explain how to carry out basic actions needed for debugging with GDB.

Using Breakpoints

Breakpoints allow execution to be halted at a specific instruction address. This is useful for inspecting program state at critical points, isolating bugs, or stepping through execution. When a breakpoint is hit, the processor stops, and control is returned to the debugger.

A breakpoint can be set from GDB using:

```
break *0x1000474
```

When the breakpoint is hit, the target halts.

Important: The breakpoint should be deleted before continuing execution:

```
delete breakpoints
continue
```

Leaving an active breakpoint may cause the core to become unresponsive.

A typical breakpoint flow is:

```
break *<addr>
continue
# execution halts at breakpoint
delete breakpoints
continue
```

Read/Write Registers

The debugger provides direct access to the CPU register file. Register values can be inspected to understand the current execution state and modified to alter program behavior.

Registers can be read directly from GDB:

```
info registers
```

Or, for a specific register:

```
print $<register_name>
```

Registers can be written using:

```
set $<register_name> = <value>
```

For example, to set the program counter:

```
set $pc = 0x01000080
```

Perform Memory Access

Memory can be examined and modified at arbitrary addresses. This enables inspection of data structures, verification of program output, and manual patching of values, which can provide valuable insight into how the program is executing at the memory level.

Memory can be read and written directly from GDB.

To read memory, use the `x` (examine) command:

```
x/<count><format> <address>
```

For examples:

```
x/4x 0x10000000     # Read 4 words in hexadecimal
x/16b 0x10000000    # Read 16 bytes
x/1i 0x10000000     # Disassemble instruction at address
```

Common format specifiers include:

- `x`: hexadecimal
- `d`: decimal
- `i`: instruction
- `b`: byte
- `h`: halfword (16-bit)

- `w`: word (32-bit)

To write to memory, use the `set` command:

```
set {<type>}<address> = <value>
```

For example:

```
set {int}0x10000000 = 0xdeadbeef
set {char}0x10000000 = 0x12
```

CHAPTER

SEVEN

SYNTHESIS AND OPTIMIZATION[1]

This chapter explains the basics of logic synthesis and optimization and offers language-agnostic guidelines on writing synthesizable RTL code. Synthesis is an essential step in the ASIC design process between the RTL design and physical implementation. It is in the beginning of the ASIC toolflow later described in the chapters *Subsystem Layout* and *SoC Layout and Subsystem Integration*.

7.1 Synthesis

Logic synthesis is a central step in the ASIC design flow. It transforms a high-level hardware description, typically written in either VHDL or Verilog, into a gate-level representation composed of the target technology's standard cells. By allowing designers to express abstract behavior and letting the tools systematically refine it into a netlist that is implemented on silicon, logic synthesis is a bridge between functional design and physical implementation.

7.1.1 Inputs and Outputs

Logic synthesis takes the RTL (register transfer level) description of a design and produces a gate-level netlist. Fig. 7.1 shows a simplified synthesis flow with the required input and minimum output.

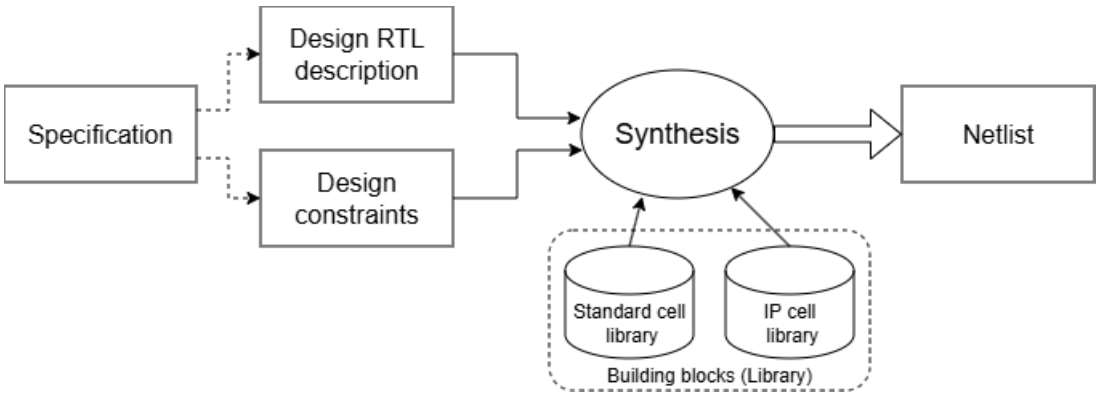

Fig. 7.1: A synthesis flow.

The synthesis tool relies on several classes of input. The first is the RTL description, which the tool elaborates to resolve hierarchy, types, generate statements, and parameterizations. The second is the collection of technology libraries, which supply logical models, timing arcs, physical abstractions, routing information, and parasitic parameters. Liberty standard-cell timing files (.lib), LEF abstracts, DEF physical data, and technology LEF routing definitions together describe the building blocks that the synthesis tool may use. A *physical-aware synthesis* flow is able to use the physical information for more accurate estimations, but for most synthesis runs providing the timing information is sufficient. Pre-placed

[1] AI tools in combination with careful proofereading and rewriting were partially used to support the writing of this section.

macros, such as memories and mixed-signal components, provide their own set of library files. These correspond to items in the IP cell library in the Fig. 7.1.

A third essential input consists of design constraints. These constraints specify clock characteristics, timing budgets, input and output delay margins, and environmental parameters such as process-voltage-temperature (PVT) corners.

The outputs of a synthesis run include a fully mapped gate-level netlist targeting the standard-cell library; timing, power, and area reports; refined constraint files for downstream placement and routing, and optional timing annotation files such as SDF for simulation.

Input for synthesis

- RTL description
- SDC: design constraints
- Library definitions:
  - Liberty (.lib): timing, power and area associated with the standard cells
  - *DEF (Design Exchange Format): physical information of the design*
  - *LEF (Library Exchange Format): abstract of the standard cells*
    - *Technology LEF: metal layer information, via definition, resistance, capacitance, antenna information, etc.*
    - *Macro LEF: macro cell physical information, pin placement, etc.*

Output of synthesis

- Verilog: Gate-level netlist
- SDC: Refined constraint files
- SDF (Standard Delay Format): Timing representation for simulation
- Reports
  - Timing
  - Area
  - Power
  - QoR (Quality of Results)

7.1.2 Synthesis Flow

The synthesis process involves several interdependent steps. The steps are listed here in order of operation.

Elaboration

- In this stage, the high-level description is parsed and analyzed to understand the design structure and functionality.
- The design may be divided into modules and submodules, and hierarchical relationships are established.

- Basic syntactic and semantic checks are performed to ensure the design is valid and follows the language rules.

Generic mapping

- The elaborated design is built with generic building blocks: boolean gates, multiplexers, adders, comparators and similar.
- Technology-independent, functional model of the design.

Technology mapping

- The generic gate-level representation is mapped onto the specific cells available in the target standard cell library.
- This involves selecting appropriate gates to implement the design's logic and interconnections.
- Mapping decisions consider factors such as fan-in, load, available drive strengths, leakage, and estimated routing delay.

Optimization

- The logic is optimized to improve performance, reduce power consumption, and minimize area usage.
- Timing optimization attempts to reduce critical path delay.
- Area optimization tries to reduce cell count or encourage smaller gates.
- Power optimization reduces switching activity, clock tree load, and leakage.
- These goals often conflict, and synthesis aims to balance them rather than achieve absolute maximum performance in any one dimension.

Writing output

- The final gate-level netlist is written to a file in Verilog format.
- The refined constraint file (SDC) and a possible SDF file with delay annotation are saved.
- The synthesis tool can generate reports of power, area, timing, and quality of results. - These are the first estimates, and the accuracy is improved later in the ASIC design flow.

7.2 Timing Closure and PVT Considerations

Realistic timing modeling is part of the synthesis process. Clock uncertainties, latency, and transition times must be specified so that the synthesized netlist does not unrealistically assume a perfect clock. External devices connected to input and output cause delays and load, and these should also be modeled. These considerations prevent optimistic timing that would later fail during physical design. The timing constraints are written in SDC files, and refining the design constraints with multiple iterations is required for successful synthesis.

There are four types of paths in a synchronous circuit, as shown in Fig. 7.2 :

- input-to-register
- register-to-register
- register-to-output

- input-to-output

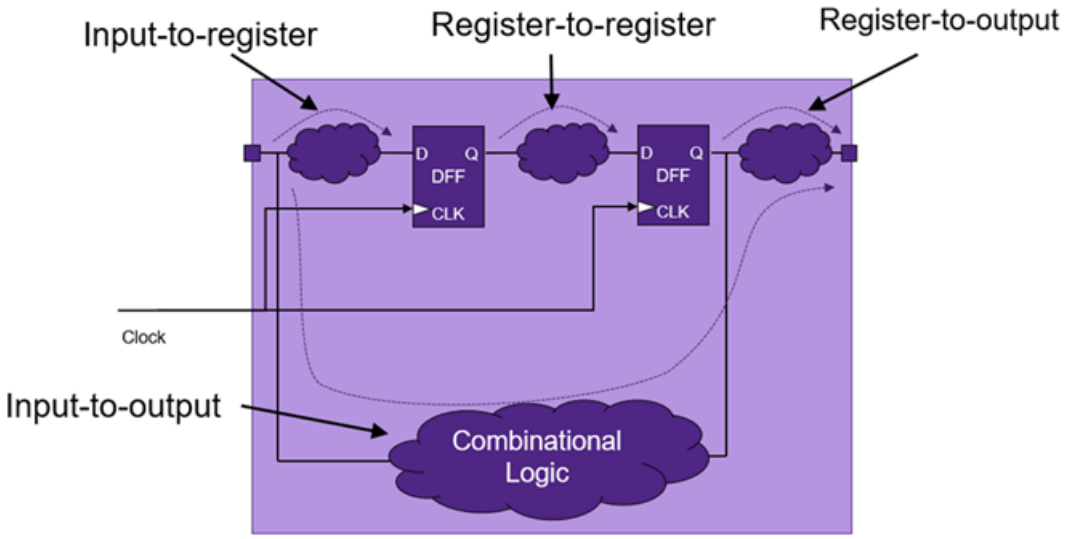

Fig. 7.2: Four types of timing paths.

The register-to-register paths are constrained by clock period and register setup/hold requirements. The delays in input and output ports must include the delays of the outer virtual logic. These are modeled with input and output delays. Pure combinational paths do not have a clock to constrain timing and an abstract clock called "virtual clock domain" should be used for constraining instead.

ASIC designs must satisfy timing across manufacturing variation, voltage fluctuations, and operating temperature ranges. Libraries specify multiple PVT (process, voltage, temperature) corners, and synthesis must consider both the slow corners used for setup checks and the fast corners relevant for hold checks. Multi-mode multi-corner (MMMC) synthesis takes not only the multiple PVT corners into account, but also the multiple operating modes of the design, such as normal operation, sleep, and test modes.

Some modern technologies exhibit temperature inversion, where colder temperatures increase delay rather than decrease it, complicating the determination of worst cases. Multi-corner synthesis ensures that no timing violations occur under any supported condition.

7.3 Optimization

Various techniques are applied to reduce complexity, enhance timing, and minimize resource usage. Some optimizations can be done manually by the designer, but usually the optimizations made by the synthesis tool lead to better results. Code clarity and ease of verification are important optimization dimensions too, and manual optimizations that sacrifice readability should be avoided.

Optimization is always a trade-off. Minimizing the critical path delay usually leads to a larger area and increased power consumption. Therefore, optimization should aim at the best balance of these dimensions. Fig. 7.3 shows an example of possible optimization points. Some of these techniques work better when optimization is done across unit boundaries, affecting the clarity of the resulting netlist. Preserving the design hierarchy can allow easier debugging but sacrifices optimization efficiency.

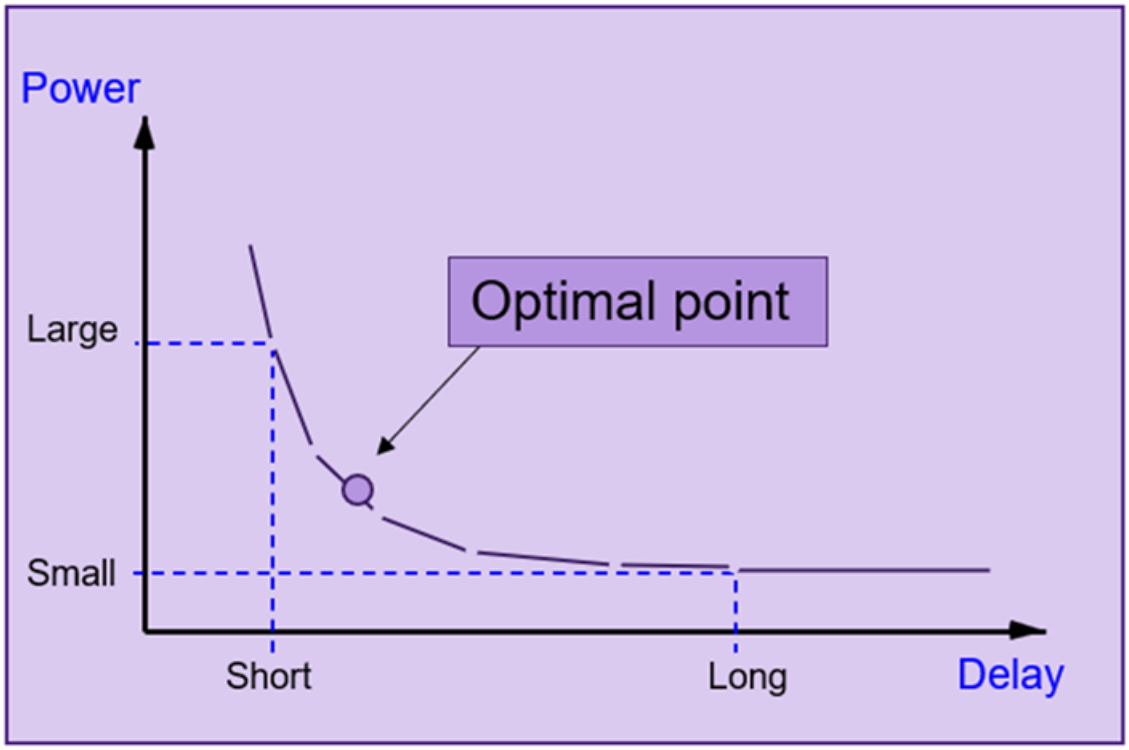

Fig. 7.3: Optimization tradeoffs.

7.3.1 Constant Propagation and Folding

Constant values are propagated through the logic, and operations involving constants are simplified or eliminated.

7.3.2 Dead-code elimination

Removing code that does not contribute to the functional output.

7.3.3 Common Subexpression Elimination

Redundant calculations are identified and merged to reduce the number of logic operations.

7.3.4 Register duplication:

Distributing load across multiple register copies, reducing routing delay in high-fan-out or long-distance paths.

7.3.5 Logic Optimization

Boolean logic minimization techniques are used to simplify logic expressions and reduce the number of gates required to implement them.

7.3.6 Resource sharing

Saving area by reusing operators like adders or multipliers when operations are mutually exclusive, inserting multiplexers on the operators' inputs or outputs.

7.3.7 Register Retiming

Flip-flops are moved along critical paths to balance delay, optimize clock frequencies, and improve performance.

7.4 Coding Style and Synthesizable Constructs

Well-structured RTL is essential for predictable hardware. Tools can also behave differently and the same structure can be interpreted in a different way between simulation and synthesis, or between different synthesis tools. Coding guidelines decrease the possibility of wrong interpretations. This is a summary of commonly described dos and don'ts related to synthesis.

Combinational processes must be free of incomplete assignments and must include all relevant signals in their sensitivity lists. Every signal must be assigned a value in every combinational process invocation. Do not instantiate basic gates (AND, NOR, ..) and remember that synthesis tools are very good at boolean logic optimization. You should aim for code clarity instead of overoptimization.

Sequential processes should be edge-driven and include, if needed, a properly separated asynchronous reset for initialization only. Initialization values, delay annotations, multiple wait statements, file operations, and unbounded loops belong in testbenches rather than synthesizable RTL.

Conditional structures map directly to multiplexers. If-elsif chains create priority networks, while case statements produce balanced trees with uniform delays. Avoiding unnecessary priority networks can significantly improve timing. Multiplexers can dominate area in certain datapaths, so designers should minimize branching logic where possible.

Loops must have static bounds to be synthesizable; static loops are unrolled into parallel logic, which may dramatically increase area. When sequential behavior is desired, an FSM or pipelined structure should be introduced. Combinational loops are illegal, and unintended latches often signal improper RTL structure.

Arithmetic operators are fundamental components of ASIC datapaths. Comparisons should often be implemented through subtraction and examination of the result's sign or zero status rather than through gate-level comparators. Shifters that support arbitrary shift amounts require substantial multiplexer networks; for a wide datapath, such as 32 bits, dynamic shifting may require hundreds or thousands of multiplexing elements. Designers may use architectural methods to reduce the need for fully dynamic shifters.

7.5 Design for Testability (DFT)

To ensure that manufactured ASICs can be tested for fabrication defects, the synthesized design should support design-for-test (DFT). The primary method of adding DFT is *scan insertion*, where normal flip-flops are replaced with scan-capable variants connected into *scan chains*. Shifting of the scan chain can be controlled to verify all the flip-flops internal circuits are without defects. Automatic test pattern generation (ATPG) tools analyze the gate-level netlist and generate test patters for improved fault detection. Self-test circuit insertion is also possible for memories (MBIST) and logic (LBIST).

CHAPTER

EIGHT

SUBSYSTEM LAYOUT[1]

8.1 Block-level layout and subsystem integration

This section describes a vendor-agnostic procedure for the block-level ASIC layout process. The intended audience are students working on a submodule developed for the Didactic-SoC of the Edu4Chip project. At this point, the students are expected to have the RTL for their submodules readily implemented. I.e., the following assumes that an RTL submodule is available, that an area budget for the block has been defined (most likely this is a specification provided by the supervisors), that timing targets for a local clock, APB and PMOD interfaces are provided (likely based on another requirement by the supervisors), and that the maximum metal layer to be used is given (a specification derived from the targeted technology). The expected final output after completing all steps is a placed-and-routed block with an extracted gate-level netlist and timing reports suitable for sign-off handoff.

8.2 Inputs and constraints

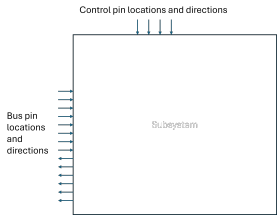

Fig. 8.1: Visualization of the Pin Location Map (no coordinates are given for this generic example but must be added for actual specification).

Make sure that the following inputs are available before starting to work on the layout:

- RTL source for the submodule (*design.v* or equivalent, likely with several submodules, etc.).
- An area budget and aspect ratio for the block.
- Timing targets and constraints for the local clock domain and for external interfaces (APB/PMOD), including target frequencies and acceptable setup/hold margins.
- Pin location map like in Fig. 8.1 for the block boundaries (fixed pins; no I/O ring or I/O cells are provided within the block).
- Power-rail definition (e.g., where are the power rails on a chip level, which is needed to design the internal power mesh) and voltage windows (minimum/maximum voltage to be accepted) for the block.

[1] AI tools in combination with careful proofereading and rewriting were partially used to support the writing of this and the following sections.

- Maximum metal layer allowed and any routing layer constraints or process restrictions.
- Standard-cell library LEF and Liberty (.lib) and any macro LEF files for memories and analog blocks (if present).

8.3 Synthesis and technology mapping

Logic synthesis converts the functional RTL description into a technology-dependent gate-level netlist. The process starts with reading the RTL and the target standard-cell library (Liberty *.lib* timing models and LEF physical site definitions). During synthesis, the following high-level optimizations are applied:

- Constant propagation,
- Dead-code elimination,
- resource sharing,
- retiming and register balancing.

Constraint information (clock definitions, false paths, multi-cycle paths) is applied so that the generated logic meets the intended timing and functionality if possible.

Technology mapping is performed after logical optimization. This step selects specific standard cells to implement the optimized logic. The implemented mapping considers drive strength, leakage and area tradeoffs, and seeks to satisfy timing and area budgets by applying optimization startegies. To achieve this goal, multi-output cells, complex gates and memory primitives from the target library may be utilized during mapping to improve density and/or timing.

The primary outputs of the synthesis step are a gate-level netlist (called, e.g., *design.gate.v*), an updated constraint file aligned with the netlist, and reports summarizing area, timing, and power estimates. In order to ensure that the tools did not unintentionally modify the design (e.g., due to problematic RTL code at the input), a formal or logic equivalence check is usually performed at the end. This netlist equivalence check ensures that the the synthesized netlist is functionally equivalent to the original RTL code.

8.4 Floorplanning and macro placement

After the synthesis and before the "actual" place and rounte, the designer has to provide some "hints" to the tool in form of defining kind of a a rough high level design. More precisely, the subblock's area is partitioned to accommodate macros (memories or analog blocks). Macros should be placed first because their aspect ratios and keep-out requirements determine the usable area and the routing resources that remain. As a rule of thump, the macros should be placed such that there is cohesive space between them to place the logic gates in a later step (e.g., in the corners of the chip) Keep-out regions must be defined explicitly to prevent illegal overlaps and to reserve space for macro pin access, power straps, and DRC (design rule check) margins. Thus, these regions should be respected during placement and any local routing operations. Nevertheless it might be necessary to place small glue logic or filler cells adjacent to a macro. Local placement constraints should be applied in this case so that keep-outs are preserved, and placement legalization and spacing checks should be used to ensure adjacent cells meet DRC and routing rules, i.e., these cases must be handled with particular care. It might happen that in the overall place-and-route process congestion appears at macro edges. In this case, macro positions or keep-out extents should be adjusted iteratively to relieve the hotspot.

Power planning is part of floorplanning, too, and shall be done seperately for the individual submodules (including definition of power rails for standard cells, definition of vias, and analysis of IR drop). The routing direction of the metal layers also determines the routing direction of the power rails. The upper layers are used to route wide power straps so that power distribution is uniform and compatible with the

layer direction rules. It is assumed here that students working on the layout only work on a submodule. Therefore it is important to note that at this stage, the top-level of the block is its only connection point to the chip-level layout. The physical connection to the global power mesh is completed later during block-integration. Inside the block, via locations and power strap pitches are designed to support basic IR-drop planning.

Finally, during floorplanning, the well ties and boundary cells are planned at the block periphery and where required around macros. Also, pin locations at the block boundary are fixed. In case of designing a submodule for the Didactic-SoC, no dedicated I/O cell ring is required. The routing to the given pin locations, e.g., for PMOD and APB interfaces, must be planned accordingly for block-integration later.

8.5 Standard-cell placement

Site rows are defined based on the height of the selected standard cell libraries. The power stripes of VDD and VSS are placed in an alternating pattern so that the top and bottom rails of the standard cells connect. Note that all standard cells can be flipped horizontally to match VDD at the top rail and VSS at the bottom or vice versa. The placement tool is then used to generate an initial placement that satisfies basic placement density and congestion heuristics.

To estimate congestion, G-boxes (also referred to as global routing tiles) are used. These, are abstract spatial tiles used specifically for congestion analysis. The layout area is divided into a regular grid of G-boxes and, for each box, routing demand, available capacity and overflow are computed to quantify local routing pressure. Per-box metrics are visualized (for example as heat maps) so that hotspots where demand exceeds capacity are easily identified. G-box size is chosen relative to metal pitch and typical netlength: larger boxes are used for early, coarse planning and smaller boxes for detailed post-placement analysis.

A first timing analysis is performed for early feedback. Macros can be moved to improve this early timing estimate. Once a satisfying configuration is achieved, the placement of the standard cells is legalized and they are moved into the site rows. Where congestion is detected, placement constraints or cell spreading is applied iteratively until local routing pressure is sufficiently reduced.

8.6 Clocking and timing planning

A preliminary clock network is initialized before detailed routing. The intended clock source, target frequencies, and clock domains are taken into account. In case of the didactic SoC, timing budgets are allocated for the APB and PMOD interfaces that account for external interface timing and the internal clock tree latency.

Timing constraints are expressed in a vendor-agnostic SDC-style form and maintained as a living file. Clock definitions, false/async paths, and max/min path exceptions are recorded.

Clock-tree synthesis (CTS) is performed to create a balanced distribution network that delivers the clock signal to all synchronous elements with controlled insertion delay and minimized skew. The CTS process inserts clock buffers and builds a hierarchical tree of buffer stages or balanced branches so that path delays from the clock root to leaf flops are matched within the required skew budget. Clock buffers (special buffered cells or dedicated cells from the standard-cell library) are used to drive the capacitive load of clock nets. Buffer sizing and staging are chosen to trade off drive strength, area and power. During CTS, clock gating cells are normally preserved at the logical level and their placement is coordinated so that gating does not introduce excessive skew or hold-time violations.

After CTS, a clock-netlist is produced that lists inserted buffers and their connections; this netlist is to be used by subsequent routing and STA (Static Timing Analysis) steps. Balancing is achieved by adjusting

buffer placement, wire routing and buffer sizing so that the worst-case difference between clock arrival times at sequential elements is within the specified skew margin.

8.7 Routing strategy

Routing is performed in two major phases: global routing, which allocates coarse routing resources and identifies congested regions, and detailed routing, which resolves exact tracks, vias and metal geometries. In this flow, clock and power distribution are assumed to have been established earlier; signal routing is therefore performed with awareness of the existing power and clock structures so that conflicts are avoided.

Global routing produces guides for long nets and buses (for example APB or PMOD) and generates an early view of resource utilization. Preferred metal layers and track grids are assigned per net class while respecting layer directionality and the maximum metal layer constraint. Timing-critical nets and matched buses are routed under constraint directives (wire-length matching, shielding, differential ordering) ahead of less critical signals so that the most timing-sensitive topology is preserved.

Congestion is monitored using the earlier described G-box analysis and global routing results; hotspots are identified visually and numerically and are addressed by placement spreading, track reservation or reassignment of layer usage.

Detailed routing then performs exact track reservation, via insertion and local jog adjustments using DRC-aware rules so that the generated geometries meet manufacturability requirements. Shielding is applied where coupling or signal integrity is a concern, and via planning follows electromigration and reliability guidance, including the use of stacked vias or via arrays when larger conduction area is required.

Once critical topology has been finalized, nets with timing constraints are locked while non-critical nets remain available for congestion relief iterations. Metal-fill and density rules are planned and applied prior to final extraction because fill geometry affects parasitics; a repeat extraction run is therefore expected. Routing DRC fixes are minimised by using DRC-aware routing and by iterating placement and routing with early DRC checks.

Via planning must consider the maximum metal layer allowed for the block; via arrays or stacked vias should be used where robustness or current density requirements demand larger conduction area. Routing congestion hotspots are to be resolved by re-placement or rerouting of non-critical nets as required.

8.8 Extraction and parasitics

Parasitic extraction converts the routed layout geometry into an electrical representation of interconnect parasitics (resistance, capacitance and, where required, cross coupling). Extraction tools read the layout, the layer stack and the routing topology and produce files such as SPEF/RCX that quantify per-net resistance and capacitance. These parasitic values are essential inputs to static timing analysis (STA) and to the generation of Standard Delay Format (SDF) timing annotations for gate-level simulation: without extracted parasitics, timing and crosstalk effects present after routing cannot be assessed accurately.

The level of extraction accuracy is chosen according to the stage of the flow and the available compute resources: coarse extraction may guide early timing decisions, while full extraction (including coupling capacitances) is required for final sign-off. Extraction is repeated after major routing changes (for example after detailed routing, metal fill insertion, or significant ECOs) so that STA and SDF back-annotation reflect the current physical design. Care is also taken to preserve instance and net naming so that the extracted data maps cleanly back to the gate-level netlist used by simulation and timing tools.

8.9 Timing checks

Static timing analysis is executed using the extracted parasitics for all relevant PVT corners. Slack reports, netlists of worst paths and cross-sectional timing tables must be inspected by the designers. Timing violations may occure at this point. They may be addressed by iterative constraint tuning, buffering, or selective cell upsizing. *Timing closure is to be demonstrated for the block's critical paths before handoff.*

8.10 Sign-Off checks

Sign-off checks such as DRC and LVS checking are performed with appropriate tools. Successful design rule checks (DRC) demonstrates that the design indeed fulfills the requirements to be manufactured. Layout versus Scematic (LVS) checks ensure that the final layouted design matches the schematic netlist before layout. If both tests succeed, the block layout can be handed over to the block integration. The sign-off steps are explained in detail in later sections.

8.11 Flow summary

The block-level flow is represented succinctly as follows (RTL → sign-off):

- RTL and testbench verification
- Synthesis → *design.gate.v*
- Floorplan + macro placement
- Power planning and routing
- Placement of standard cells
- Clock tree synthesis (CTS)
- Global and detailed routing and via insertion
- Parasitic extraction (SPEF/RCX) and SDF generation
- STA across PVT corners and GLS runs on reduced regression
- DRC/LVS and final report generation

8.12 Deliverables and handoff

The following minimum deliverables are provided for block handoff:

- RTL snapshot and synthesis scripts.
- Constraint file(s) (SDC placeholders) describing clocks and interface timing.
- Floorplan description and LEF/DEF placeholders (or actual LEF/DEF if available).
- Placed and routed database or exported DEF and GDS/LEF as applicable.
- Extracted parasitic file (SPEF/RCX) and SDF timing annotations.
- Gate-level netlist (*design.gate.v*) and selected annotated GLS waveforms.
- STA reports for required corners and PVTs.
- DRC and LVS reports demonstrating clean results.
- Test cases for simulations to demonstrate functional correctness.

8.13 Example SDC snippets

The following SDC-style snippets are provided as vendor-agnostic placeholders. They are adapted to the chosen toolchain and PDK. All names in angle brackets (<...>) denote variables to be replaced.

Define clocks and input/output delays:

```
create_clock -name clk_core -period <CLK_PERIOD_NS> [get_ports <CLOCK_PIN>]
# APB interface timing
create_clock -name clk_bus -period <BUS_CLK_PERIOD_NS> [get_ports <APB_CLOCK_
↪PIN>]
set_input_delay -clock clk_bus <INPUT_DELAY> [get_ports <APB_INPUT_PINS>]
set_output_delay -clock clk_bus <OUTPUT_DELAY> [get_ports <APB_OUTPUT_PINS>]
```

Declare false paths and async paths:

```
# asynchronous reset paths
set_false_path -from [get_ports <ASYNC_RESET_PIN>]
```

8.14 Short LEF and Liberty examples (illustrative)

The following minimal examples are provided as educational placeholders. They are intentionally compact; vendor or PDK files contain much more detail and must be used for real P&R flows.

Minimal LEF snippet:

```
MACRO AND2
  CLASS CORE ;
  SIZE 2.0 BY 1.2 ;
  PIN A
    DIRECTION INPUT ;
    PORT
      LAYER M1 ; RECT 0.10 0.10 0.30 0.30 ;
    END
  END A
  PIN B
    DIRECTION INPUT ;
    PORT
      LAYER M1 ; RECT 0.40 0.10 0.60 0.30 ;
    END
  END B
  PIN Y
    DIRECTION OUTPUT ;
    PORT
      LAYER M1 ; RECT 1.70 0.10 1.90 0.30 ;
    END
  END Y
END AND2
```

Minimal Liberty snippet:

```
library (AND_lib) {
  cell (AND2) {
    area : 1.2;
    function : "Y = A & B";            /* Boolean function */
    cell_leakage : 1.2e-6;             /* leakage in Watts (example) */
    pin (A) { direction : input; capacitance : 0.02; }
    pin (B) { direction : input; capacitance : 0.02; }
    pin (Y) { direction : output; capacitance : 0.01; }

    /* Simple 3x3 cell_rise timing table:
       index_1 = input transition (ns) -> rows
       index_2 = output load (fF)       -> columns
       values are row-major: for each index_1 row, values for each index_2␣
↪column follow.
    */
    cell_rise (
      index_1 (0.01 0.05 0.10)    /* input slew (ns) */
      index_2 (1.0  2.0  4.0)     /* output load (fF) */
      values (
        0.10 0.12 0.15
        0.11 0.13 0.16
        0.12 0.14 0.17
      )
    )
  }
}
```

CHAPTER

NINE

SOC LAYOUT AND SUBSYSTEM INTEGRATION

The ASIC implementation flow described in this chapter follows directly from the previous section: the pipeline of RTL refinement, synthesis, floorplanning, power planning, clock-tree synthesis, place & route and final signoff remains the same. The principal addition at this stage is a focused activity around IO-ring planning, which we will discussed in detail later. IO-ring planning is an early floorplanning and padframe task that fixes the die outline, determines pad types and pitches. Because it establishes physical anchors for off-chip interfaces, it has a direct influence on placement, routing and power network design.

Subsystems are integrated in the same way as other large macros. Each subsystem is introduced with a defined physical footprint, pin map and placement constraints. Integration means placing the macro in accordance with the global floorplan, connecting its power pins to the global power-rail scheme, and aligning its signal pins to routing channels so that routing access is preserved. After placement, incremental timing analysis, DRC/LVS checks and power-integrity analyses are run to validate the integration and to reveal any conflicts or timing violations.

9.1 IO Planning

IO planning starts with the selection of the appropriate I/O cell variants from the technology library, matching each interface's voltage domains, signaling modality (single-ended or differential), termination and ESD requirements. Examine the RTL code classify the nets - high-speed lanes, memory interfaces, control and test ports - so the signal pad budget and pin-grouping constraints are known before physical planning. As a rule-of-thumb for initial sizing, provision roughly fifty percent additional pad resources for power/ground and power-delivery network (PDN) structures relative to the signal pad count. This reserve accommodates separate IO power domains, robust ground paths and decoupling, reduces PDN impedance, and mitigates simultaneous-switching noise. Position the selected I/O cells on the padframe, inserting corner and filler cells and enforcing orientation and mechanical constraints so the padframe is manufacturable and LVS-clean. Finally, integrate the padframe with the die by tying I/O power pins into the on-die power mesh and routing internal signal connections to the core. Then validate the assembly through functional simulations and IR drop analysis. A sketch of an IO ring is shown in Fig. 9.1; the pinout used in the tapeout of the first revision of the didactic SoC is shown in Fig. 9.2. It is given as an example.

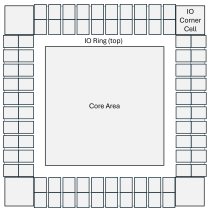

Fig. 9.1: Visualization of an IO Ring.

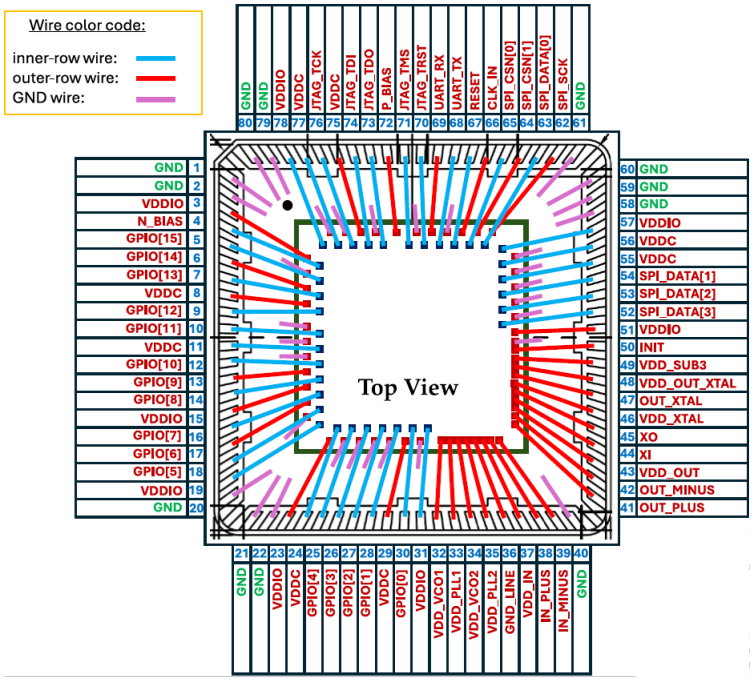

Fig. 9.2: Example pinout for the Didactic-SoC ASIC.

9.2 Chip-level Timing Constraints

In addition to the previously described constraints for clocks and reset paths, chip-level timing constraints must explicitly capture I/O timing arcs and the board/package contributions that affect them. External device characteristics (for example, a USB-to-JTAG bridge) are modeled with measured or vendor-supplied delays and IBIS/S-parameter data so that input/output delays, setup/hold margins and jitter budgets are expressed in the chip's SDC files. Constraints should therefore include create_clock statements, pin-level input/output delays referenced to PCB paths, and any timing exceptions (false paths or multicycle paths) together with agreed signoff margins. These constraints must be iterated with padframe and pin assignments because pin choice and package parasitics materially affect timing closure and signoff results.

CHAPTER

TEN

IC VALIDATION AND SIGN-OFF

This section summarizes core concepts, a compact verification flow, and practical checklists for validating an integrated circuit and preparing it for sign-off.

10.1 Overview & Goals

Validation and sign-off ensure the design is functionally correct after the physical design, meets timing and power constraints, and is manufacturable. In technical terms, the key goals are the functional correctness of the designed silicon, timing closure with margins under different temperature and voltage corners, physical correctness (DRC/LVS), and the preparation of deliverables for tape-out and post-silicon bring-up.

10.2 Verification Flow

The verification flow encompass the following steps:

- RTL verification
- Synthesis and backend design
- Filler insertion
- Static timing analysis (STA)
- Timing-annotated gate-level simulation
- Physical verification (DRC/LVS)
- Sign-off deliverables

The concepts of RTL verification, synthesis and backend design are described in previous sections. After the physical design, the chip area without standard cells and without wires has to be filled with dummy metal to achieve a high density avoiding large holes in the physical chip. This additional metal introduces new capacities, which impact timing. Thus, RC extraction and STA are repeated after filler insertion. STA exhaustively tests if the paths through the circuit meet the timing requirements (setup/hold, max/min paths). While the same step is applied during backend design, STA considers more details, e.g., exact RC values, and is performed for all PVT corners, which would be too computationally costly during backend design. The delays of the standard cells and wires are saved in a Standard Delay Format (SDF) file. With these delays, gate-level simulations can be annotated to capture the actual timing and check for any issues, mainly for clock-domain crossing and asynchronous resets.

10.2.1 Gate-Level Simulation (GLS)

A gate-level netlist is the structural representation produced by synthesis: it instantiates library standard cells (gates, flops, buffers, memories) and nets that connect them. Each standard cell is modeled at the RTL-verification level by a gate-level Verilog model. SDF provides per-instance timing values (cell delays and interconnect delays) that are applied to those models for timing-accurate simulation. The simulator is event-driven and evaluates many more signal transitions than RTL-level simulators, so GLS runs are significantly slower and heavier on memory.

GLS typically uses the same test vectors and directed tests that were applied during RTL verification (reset/boot sequences, basic transactions, corner functional tests), but execution is much slower due to the larger netlist and added timing events. Because of this cost, often a subset of functional checks is performed plus a few timing-specific scenarios (clock gating, edge-cases, race-prone sequences) rather than full RTL regression scope.

10.2.2 Physical verification

Design Rule Check (DRC) is an automated layout check against the foundry design rule set to ensure manufacturability. Typical DRC checks include minimum/maximum widths and spacing for metal/poly/diffusion, via/contact enclosure and overlap, minimum implant/active spacing, well and guard-ring requirements, metal density and fill constraints, antenna/etch rules, and array/abutment constraints for standard-cell placement. DRC finds geometric violations that could cause shorts, opens, yield issues, or lithography failures; fixes usually require layout edits, spacing adjustments, or metal fill changes and must be rechecked iteratively. In summary, only a DRC clean design can be manufactured reliably.

Layout Versus Schematic (LVS) is a connectivity and device-level comparison between the extracted layout netlist and the original schematic/netlist. LVS flow extracts transistors, resistors, capacitors, and nets from the layout (including stacked/series devices and device sizing) and reports mismatches such as missing or extra devices, wrong connections, pin mapping errors, and net name/driver conflicts. LVS ensures the implemented layout is electrically equivalent to the intended design; typical iterations fix pin placements, routed nets, or device stacking/sizing errors.

DRC/LVS-clean reports are often required sign-off artifacts. They ensure the correct PDK/DRC decks and LVS mappings are used and include the generated reports in the deliverables.

10.3 Sign-Off Checklist

- Functional verification: targeted regression passing, sanity coverage for critical IP.
- Timing: STA reports for target process and PVT corners; no unresolved setup/hold failures.
- Physical: DRC/LVS clean for whole design.
- Power/IR: basic power estimation and hotspots identified.
- Testability: scan chains or JTAG accessible; basic test vectors for bring-up documented.
- Deliverables: RTL snapshot, synthesis netlist, SDF, STA reports, DRC/LVS reports, test vectors.

10.4 JTAG & Silicon Bring-Up

JTAG can be used at bring-up to run boundary-scan structural tests. In this project, it provides a low-level access path into the chip for loading test patterns, reading status, and programming on-chip debug blocks. It follows the RTL implementation and the chip can be used the same way as an FPGA is. However, for

initial silicon validation, a concise bring-up plan has to list the physical pinout, the power-rail sequencing and tolerated voltage windows, and the reset/assertion timing expected by the design.

Start bring-up with quick structural checks such as verifying the IDCODEs. A set of small tests follows that exercise basic functionality (clocking, reset release, UART or console heartbeat, simple memory read/write or BIST where available). The same RTL regressions can be executed via JTAG to confirm functionality. These are typically short, deterministic tests that isolate subsystems. After that, the chip can be handed off to the software and system testing with large complex workloads.

CHAPTER

ELEVEN

OPEN SOURCE EDA TOOLS

This chapter is a summary of an open-source chip design book [Sch25], which is part of the course at DTU on introduction into chip design[1] that uses open-source tools only and where all teaching material is available in open source. In the spring semester of 2026, we design an open-source chip that will be taped out with the MPW schedule on SkyWater 130 nm.[2]

11.1 Frontend Tools

11.1.1 Chisel

Chisel is a hardware construction language embedded in Scala [BVR+12] [Sch19]. As Chisel is embedded in Scala, Chisel allows object-oriented and functional *programming* of hardware enabling a higher level of abstraction compared to traditional HDLs. This also allows designers to write so-called hardware generators [SDP+25]. Chisel designs are compiled into synthesizable Verilog, making it compatible with existing FPGA and ASIC tool flows.

11.2 History of Design Tools

Very early chips were designed by hand, and the photo masks were produced by *drawing* them with tape. As this process does not scale, computer-aided design (CAD) tools have been developed. Another term used is electronic design automation (EDA).

Alberto Sangiovanni-Vincentelli was invited to give a keynote speech at the 40th Design Automation Conference (DAC). That speech resulted in a paper on the history (and future) of EDA [SV03], mostly work as presented at DAC. Alberto co-founded Cadence Design Systems and Synopsys, the two major EDA tool vendors.

In recent years, open-source EDA tools have gained increasing attention as an alternative to proprietary toolchains. They aim to make hardware design more accessible by removing licensing barriers and play an important role, especially in education and research.

11.3 Open-Source Production Frameworks

Chip design consists of several steps from synthesis down to GDSII files. Scripts usually orchestrate those steps. In the following, we describe different frameworks that contain the needed open-source tool, but also the scripts to run the flow.

[1] https://github.com/os-chip-design/chip-design-intro
[2] https://github.com/os-chip-design/dtu-soc-2026

11.3.1 OpenROAD

OpenROAD [ACFogacca+19] started as a DARPA-sponsored project to enable an end-to-end design flow of chips from RTL to the final chips without human intervention, within a maximum of 24 hours. The documentation is published on Read the Docs. OpenROAD includes the following open-source tools:

- ifp defines the core area, the rows, and the tracks.
- pdn is a power distribution network (PDN) generator.
- Tapcell inserts tapcells or endcaps.
- gpl performs global placement. The tool is an extension of the RePlAce tool.
- Gate Resizer to optimize the design until the maximum utilization is reached.
- OpenDP performs detailed placements.
- TritonCTS 2.0 is performs clock tree synthesis.
- FastRoute performs global routing.
- TritonRoute performs detailed routing.
- OpenRCX performs parasitics extraction.
- And some more.

Use the links for documentation of the individual tools.

11.3.2 OpenLane

OpenLane [SE20, GS20] is a collection of EDA tools, such as Yosys, Magic, Netgen, and KLayout. It also includes OpenROAD. OpenLane consists of TCL scripts to automate the flow from RTL, described in Verilog, to the chip production data as a GDSII file.

OpenLane was developed by eFabless, and the documentation is published on Read the Docs. Figure 11.1 shows the design flow of OpenLane.

11.3.3 OpenLane2 and LibreLane

OpenLane2 is a new, partly rewritten,[3] framework for chip design with a focus on substituting TCL scripts with Python scripts. The documentation is published on Read the Docs. However, the original OpenLane documentation is richer than the OpenLane2 documentation. Therefore, consider reading that one as a basis.

Efabless, the original developer of OpenLane and OpenLane2, exited the business in spring 2024. To continue the development of OpenLane, the repository was cloned/forked in April 2025, and the project was renamed to LibreLane.

11.4 Tool Installation

The magic of open-source tools is that we can install them on a personal computer without any concerns of licensing and license servers. The downside of open-source tools is the variety of installation options. There is no single best way for the installation. Furthermore, open-source tools are best supported on a Unix operating system, and preferably Linux. Support under macOS comes second. MS Windows is best served by using the Windows Subsystem for Linux.

[3] OpenLane2 started out of OpenLane and OpenROAD scripts

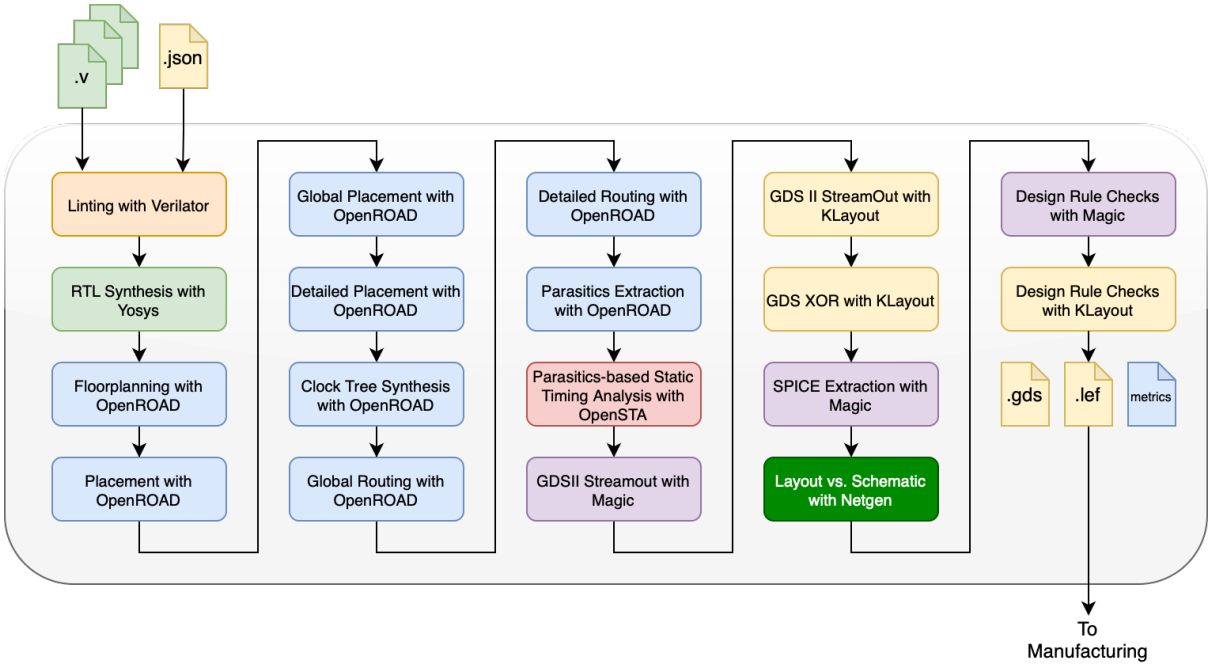

Fig. 11.1: OpenLane design flow, including the OpenROAD flow in blue. Copyright 2020-2022 Efabless Corporation and contributors, License: Apache 2.0.

11.4.1 Nix Based

From OpenLane2 (now LibreLane) onward, the preferred installation is Nix-based. Nix is a cross-platform package manager that also provides native binaries for x86 and ARM-based systems (e.g., Mac laptops). The LibreLane installation documentation explains how to install Nix, and also how to set up the Nix cache for the LibreLane tools.

Another option to install Nix is via Determinate. Note that you need to install the LibreLane cache to avoid building all the tools yourself.

11.4.2 Docker Images

Another option is to use Docker containers with ready-installed tools. Harald Prettl, from the Johannes Kepler University of Linz, provides IIC-OSIC-TOOLS (Integrated Infrastructure for Collaborative Open Source IC Tools), a Docker container based on Ubuntu 24.04 LTS for the following CPU architectures: x86_64/amd64 and aarch64/arm64.

The folder `/foss/designs` is the place to access user data on your local machine. It points to the directory pointed by the environment variable `DESIGNS`, the default is `$HOME/eda/designs`. To change this to start with your home folder, invoke the Docker image with:

```
DESIGNS=$HOME ./start_x.sh
```

Note that, as of the time of writing, the default PDK is the IHP PDK, which is not yet supported in the LibreLane flow from within the container.[4] Switch to the Sky130 PDK with the following command:

```
sak-pdk sky130A
```

[4] See: https://github.com/iic-jku/IIC-OSIC-TOOLS/issues/147

```
import librelane
print(librelane.__version__)
```

Listing 11.1: Python script to show the version of the installed LibreLane (version.py).

11.4.3 Compiling from Source

As the tools are open-source, it is always a (theoretical) option to compile them from source. However, often there are library dependencies that are not easy to resolve. Therefore, we recommend installing the tools with some packaging software that ensures compatibility between the various tools.

11.4.4 Installing LibreLane

When you use the `nix`-based setup, you need to install LibreLane. Install LibreLane by cloning it:

```
git clone git@github.com:librelane/librelane.git
```

You set up your environment by entering `nix` as follows:

```
cd librelane
nix-shell
```

The frontend command is `librelane`. Check the version you have installed with:

```
librelane --version
```

LibreLane also installs the Sky130 PDK using the `ciel` tool. The PDKs are stored under `$HOME:/.ciel`.

11.4.5 Further Packages

There are several other distributions of the open-source chip design tools available:

OSS CAD Suite is a collection of tools with a focus on open-source design flow for FPGAs.

YoWASP (Yosys WebAssembly Synthesis & PnR) is a collection of tools targeting FPGA design flow compiled to WebAssembly. Therefore, they can be easily run on different platforms, even as a Visual Studio Code plugin.

11.5 Hello World

After installing the tools, we want to explore them in action. The *Hello World* version for the backend chip design is to synthesize a minimal circuit from the HDL description to generate a GDSII file and visualize it in an editor. Note that all code examples in this document are part of the book's GitHub repository.

Execute 'nix-shell' in your LibreLane installation. Try to check your LibreLane version by invoking Python on the code shown in Listing 11.1.

Then, switch to a working directory and create the following small Verilog design (`adder.v`), as shown in Listing 11.2. This example is an adder, including registers at the input and output ports, so we can use static timing analysis (STA) to explore the maximum clocking frequency.

Furthermore, you need a configuration for your flow. You can find all possible configuration parameters at the Universal Flow Configuration Variables section of the LibreLane documentation. However, most variables can be left at their default values.

```
module adder (
    input clock,
    input  [7:0] a,
    input  [7:0] b,
    output [7:0] sum
);

    reg [7:0] reg_a, reg_b, reg_sum;

    always @(posedge clock) begin
        reg_a <= a;
        reg_b <= b;
        reg_sum <= reg_a + reg_b;
    end;

    assign sum = reg_sum;

endmodule
```

Listing 11.2: A pipelined adder as a Hello World example for LibreLane (adder.v).

```
DESIGN_NAME: adder
VERILOG_FILES: dir::adder.v
CLOCK_PERIOD: 20
CLOCK_PORT: clock
```

Listing 11.3: The YAML configuration file (adder.yaml).

Config files can support more than one PDK, see the LibreLane example. Configuration can be in YAML or JSON format. We will use the `adder.yaml` file with a minimal configuration, as shown in Listing 11.5.

For a fully automatic run of the full flow, you can execute:

```
librelane adder.yaml
```

This single command runs the complete synthesis flow from RTL to GDSII. As the design is very simple, the flow will run in about a minute. Figure 11.2 shows the final GDSII file for the adder in KLayout.

A flow takes, at the time of this writing, with LibreLane v2.4.8, 74 steps, each reporting in its folder. During the flow, a lot of information is printed out. Scrolling back in the terminal, you can see reports of design checks and resource usage. Every time you run the flow, it will create a new folder under folder `runs`. The name of the folder will be `RUN_year_month_day_hh-mm-ss`.

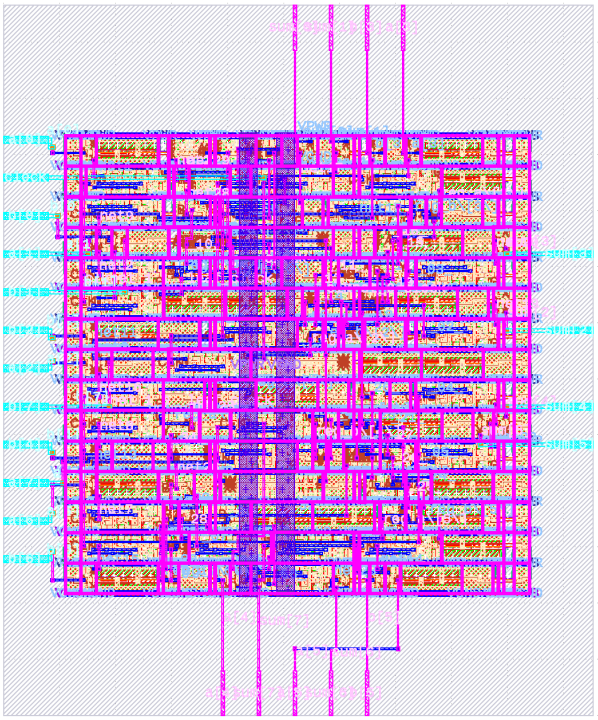

Fig. 11.2: The synthesized adder as visualized in KLayout.

11.5.1 Exploring the Design

A first step to explore the design is to view the GDS in KLayout with:

```
librelane --last-run --flow openinklayout adder.yaml
```

In Klayout, you can explore the design. Use the rulers to measure your design. How large is it? Would it fit into a Tiny Tapeout Tile? How large is a Tiny Tapeout tile? When you expand the adder in Cells, you can see which cells have been used. By double-clicking on a cell type, it is removed from the display. This is an indirect way to show the cells in the GDS. The flip-flop standard cells in Sky130 have names that include “dfxtp” in their name. Double-click that cell to see how many flip-flops are used. How many should there be?

Another option is to open the design with the OpenROAD viewer:

```
librelane --last-run --flow openinopenroad adder.yaml
```

Matt Venn has a good selection of layers and color assignments at klayout_gds.xml. You can use it with klayout as follows:

```
klayout -l klayout_gds
```

That configuration file is part of the summary tools, which can be used for a quick exploration of the design. Clone the repository, add it to your path, and set the PDK_ROOT variable (easiest into a file that you source, or add those lines to your $HOME/.bashrc):

```
export PATH=$PATH:$HOME/path/to/librelane_summary
export PDK_ROOT=$HOME/.ciel
```

Explore the tool with:

```
summary.py --help
```

When in a folder that contains the LibreLane runs, explore the GDS of the latest run with:

```
summary.py --gds
```

At the time of this writing, the flow generates 74 folders containing the results and reports for each individual step. You can find those subfolders in the folder runs/RUN_DATE_TIME. Not all reports are equally important. We will explore some of them.

Linting and a First Area Estimate

The first four steps perform linting of the design, and any Verilog syntax errors are identified during this process. An initial estimate of the used cells can be found in the report 06-yosys-synthesis/reports/stat.rpt. For our pipelined adder, we can identify the 24 flip-flops as dfxtp_2 cells in that report:

```
...
   Number of cells:                 60
     sky130_fd_sc_hd__a31oi_2        1
     sky130_fd_sc_hd__and2_2         4
     sky130_fd_sc_hd__dfxtp_2       24
...
```

We use the Sky130 PDK and can find the documentation of each cell online. E.g., the DFF is described at dfxtp. The file adder.nl.v contains the design synthesized to standard cells from the used PDK. As our example is very simple, we can manually inspect the generated Verilog.

Static Timing Analysis

In folder nn-openroad-stapostpnr, the static timing analysis (STA) reports are found.

Size

When not constraining the chip's size, LibreLane will choose the size.

Summary

The subfolder final contains a summary of the project in files metrics.csv and metrics.json. It includes timing information, size of the design, number of standard cells used, and other detailed metrics.

```
import ciel
from ciel.source import StaticWebDataSource
from librelane.common import get_opdks_rev, ScopedFile

ciel.enable(
    ciel.get_ciel_home(),
    "sky130",
    get_opdks_rev(),
    data_source =
        StaticWebDataSource("https://fossi-foundation.github.io/ciel-releases"),
)
```

Listing 11.4: Setting up the PDK (pdk.py).

11.5.2 Change the Design

The current example has non-initialized registers after power-up. Although those random values will be cleared with valid values after on and two clock cycles, it is good practise to reset registers. This expcially useful for verification.

There are two ways flip-flops can be reset: (1) with an asynchronous reset or (2) with a synchronous reset. Explain which one is to be preferred and why. Which one needs a larger area? This question is not easy to explain by a back-of-the-envelope calculation when we do (yet) not know the available standard cells and their sizes.

However, we can easily decide this question by exploring both versions with SkyWater130. Change the design to reset all registers. Run both versions through the flow. Then explore the size and which flip-flop types have been used.

11.6 Manual Flow with Python

However, we can also run the individual steps from Python. We will reuse the pipelined adder for this exploration from Section 11.5. The following example is inspired by running LibreLane on Google Colab, a nice way to explore LibreLane just from your browser.

All code from this document can be found in the GitHub repo of the book [Sch25]: code.

Start Python from within your Nix shell,

```
python
```

and run the following commands. Import LibreLane and print its version.

```
import librelane
print(librelane.__version__)
```

11.6.1 Configuration

To run our steps we need to configure the PDK via `ciel`[5] and configure our project.

Execute the code from Listing 11.4 to setup the PDK. If you have downloaded the code from the book [Sch25], you can execute the script from within Python with the following command:

```
exec(open("pdk.py").read())
```

[5] `ciel` is a PDK managemant tool, part of LibreLane.

```
from librelane.config import Config

Config.interactive(
    "adder",
    PDK = "sky130A",
    CLOCK_PORT = "clock",
    CLOCK_NET = "clock",
    CLOCK_PERIOD = 20,
    PRIMARY_GDSII_STREAMOUT_TOOL = "klayout",
)
```

Listing 11.5: Configure the project (`config.py`).

```
from librelane.steps import Step
from librelane.state import State

initial_state = State()

Synthesis = Step.factory.get("Yosys.Synthesis")
Synthesis.display_help()
```

Listing 11.6: Get started with the steps (`steps.py`).

A LibreLane `Flow` needs to be configured. Execute Listing 11.5 to configure your project. The configuration is similar to the YAML file we used in the initial example.

The results of the following steps will be placed into the folder `librelane_run`.

11.6.2 Synthesis with Yosys

The design is split into so-called *steps*. Therefore, we need to import `Step` and `State`, as shown in Listing 11.6). The last command shows us help with the steps.

We start with the Yosys synthesis. Yosys converts high-level Verilog to a low-level Verilog that includes instances of standard cells. We pass the Verilog files to `Yosys.Synthesis`. We can run the synthesis with the code from Listing 11.7.

Explore the reports and the generated low-level Verilog in folder `1-yosys-synthesis`. Furthermore, Yosys also generates visualization in the form of DOT graphs. Install Graphviz to convert those files to PDF:

```
dot -Tpdf hierarchy.dot -o hierarchy.pdf
```

11.6.3 Floorplanning

The next step is floorplanning. It determines the size of the chip and generates a cell placement grid. Each cell in the grid is called a *site*. A standard cell usually occupies multiple sites by width (e.g., a D-FF has a larger width than a NAND gate). We run the floorplanning with the code from Listing 11.8. Explore the reports in folder `2-openroad-floorplan`.

```
synthesis = Synthesis(state_in=initial_state, VERILOG_FILES=["adder.v"])
synthesis.start()
```

Listing 11.7: Running the synthesis (`synth.py`).

```
Floorplan = Step.factory.get("OpenROAD.Floorplan")

floorplan = Floorplan(state_in=synthesis.state_out)
floorplan.start()
```

Listing 11.8: Running the floorplanning (floor.py).

```
TapEndcapInsertion = Step.factory.get("OpenROAD.TapEndcapInsertion")

tdi = TapEndcapInsertion(state_in=floorplan.state_out)
tdi.start()
```

Listing 11.9: Running the tap cell insertation (tap.py).

11.6.4 Tap/Endcap Cell Insertion

This step places endcap cells and tap cells. Endcap cells are placed at the beginning and end of each row. They terminate each power line.

The tap cells connect VDD to the n-well and GND to the p-substrate. To save area, the standard cells do not connect to the tap themself. The maximum distance between tap cells is dependent on the foundry process. Run the tap placements with the code from Listing 11.9.

To view the result, start the OpenRoad GUI with:

```
openroad -gui
```

and then open `adder.odb` from folder `3-openroad-tapendcapinsertion`.

11.6.5 I/O Placement

Metal pins are placed at the edges of the design for the top-level inputs and outputs. The default placement can be overridden by a placement configuration.

Run the tap placements with the code from Listing 11.10 and explore the result with the OpenRoad GUI.

11.6.6 Power Distribution Network (PDN)

The next step is to generate the power distribution network (PDN). It is a pattern of vertical and horizontal straps. The vertical straps are thicker and distribute power to the horizontal straps. The horizontal straps are delivering power to the individual standard cells. More details of the configuration of the PDN can be found in the LibreLane documentation.

Run the PDN generation with the code from Listing 11.11 and explore the result with the OpenRoad GUI. Figure 11.3 shows the PDN of our small adder example.

```
IOPlacement = Step.factory.get("OpenROAD.IOPlacement")

ioplace = IOPlacement(state_in=tdi.state_out)
ioplace.start()
```

Listing 11.10: Running the I/O placement (io.py).

```
GeneratePDN = Step.factory.get("OpenROAD.GeneratePDN")

pdn = GeneratePDN(
    state_in=ioplace.state_out
)
pdn.start()
```

Listing 11.11: Running PDN generation (pdn.py).

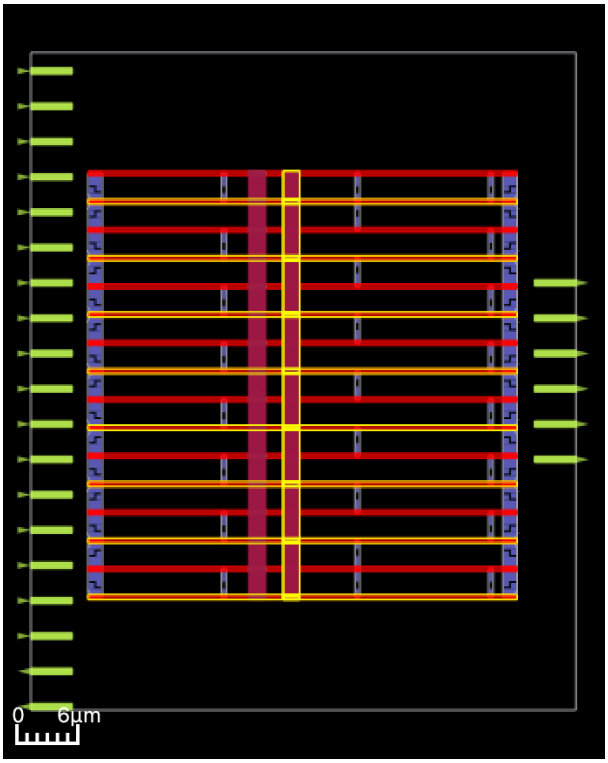

Fig. 11.3: Chip layout with pins and power distribution network.

11.6.7 Global Placement

Global placement places the standard cells, aiming to minimize the distance between connected cells. The cells are not exactly placed, as we can see in Figure 11.4. The placement is even *illegal* as it is not properly aligned with the grid. This is fixed in the next step of detailed placement, also called placement legalization.

Run the global placement with the code from Listing 11.12 and explore the result with the OpenRoad GUI.

```
GlobalPlacement = Step.factory.get("OpenROAD.GlobalPlacement")

gpl = GlobalPlacement(state_in=pdn.state_out)
gpl.start()
```

Listing 11.12: Global placement (glp.py).

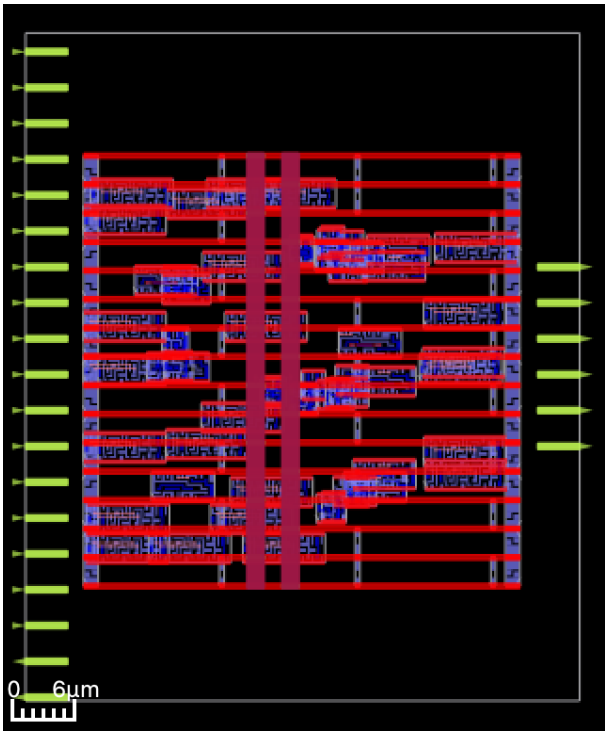

Fig. 11.4: Global placement.

```
DetailedPlacement = Step.factory.get("OpenROAD.DetailedPlacement")

dpl = DetailedPlacement(state_in=gpl.state_out)
dpl.start()
```

Listing 11.13: Local placement (`dlp.py`).

11.6.8 Detailed Placement

Detailed placement aligns the cells with the grid; we call this also *legalizing* it. Run the detailed placement with the code from Listing 11.13 and explore the result with the OpenRoad GUI. Figure 11.5 shows the now well-aligned cells.

11.6.9 Clock Tree Synthesis (CTS)

After placing all cells, we create the clock tree, including buffers for the clock. Run the detailed placement with the code from Listing 11.14.

```
CTS = Step.factory.get("OpenROAD.CTS")

cts = CTS(state_in=dpl.state_out)
cts.start()
```

Listing 11.14: Local placement (`cts.py`).

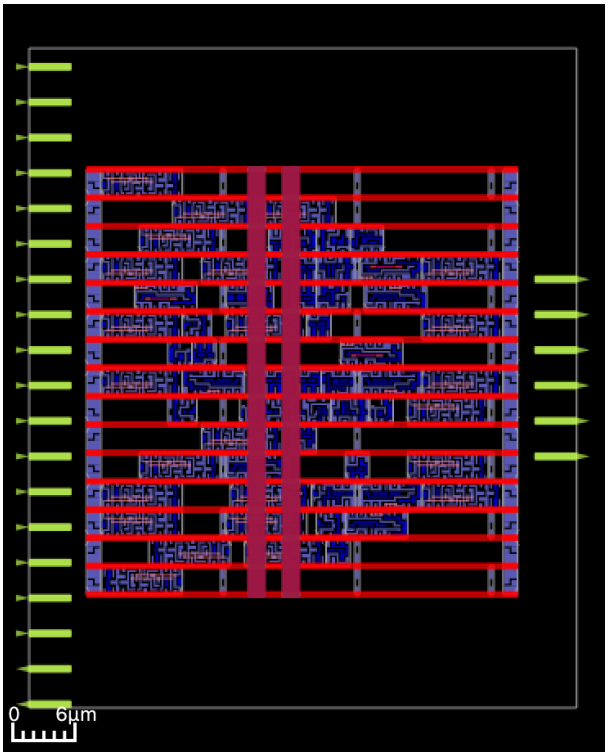

Fig. 11.5: Legalized placement

```
GlobalRouting = Step.factory.get("OpenROAD.GlobalRouting")

grt = GlobalRouting(state_in=cts.state_out)
grt.start()

DetailedRouting = Step.factory.get("OpenROAD.DetailedRouting")

drt = DetailedRouting(state_in=grt.state_out)
drt.start()
```

Listing 11.15: Global and local routing (`route.py`).

11.6.10 Global and Detailed Routing

As the result of global routing is stored in an internal data structure, we see no effect on the actual design. Therefore, we combine global routing and local routing in a single Python script to be executed (Listing 11.15). Figure 11.5 shows the routed design.

11.6.11 Fill Insertion

Fill insertion fills the gaps between the placed cells. Execute Listing 11.16 and explore the output with the OpenRoad GUI.

11.6.12 Resistance/Capacitance Extraction (RCX)

As preparation for timing analysis, the resistance and capacitance needs to be extrated. Execute Listing 11.17 to run this step.

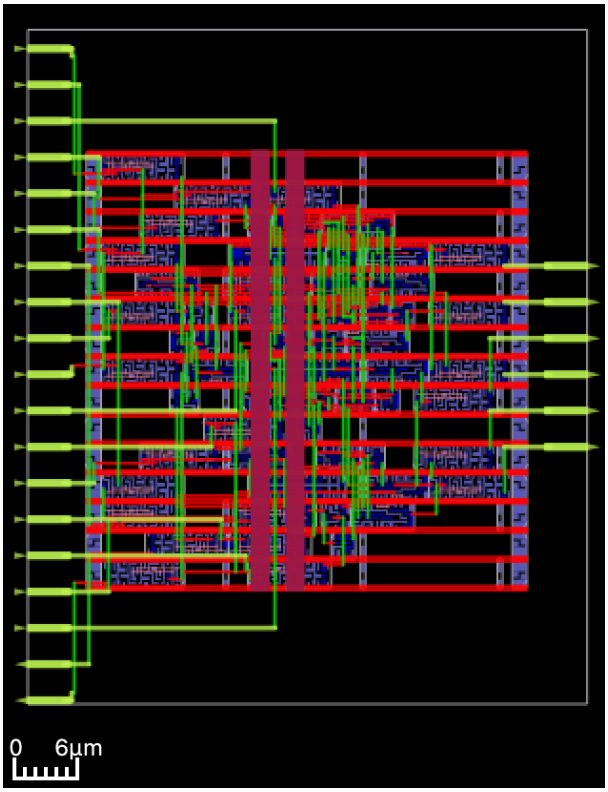

Fig. 11.6: The routed design.

```
FillInsertion = Step.factory.get("OpenROAD.FillInsertion")

fill = FillInsertion(state_in=drt.state_out)
fill.start()
```

Listing 11.16: Fill insertion (`fill.py`).

11.6.13 Static Timing Analysis (STA)

Tu run Static Timing Analysis, execute Listing 11.18.

11.6.14 Generating the GDSII

Finally, we can stream out the GDSII of our design to ship for manufacturing.

Execute Listing 11.19. It is now again time to explore the final result with KLayout

```
RCX = Step.factory.get("OpenROAD.RCX")

rcx = RCX(state_in=fill.state_out)
rcx.start()
```

Listing 11.17: Fill insertion (`rcx.py`).

```
STAPostPNR = Step.factory.get("OpenROAD.STAPostPNR")

sta_post_pnr = STAPostPNR(state_in=rcx.state_out)
sta_post_pnr.start()
```

Listing 11.18: Static timing analysis (sta.py).

```
StreamOut = Step.factory.get("KLayout.StreamOut")

gds = StreamOut(state_in=sta_post_pnr.state_out)
gds.start()
```

Listing 11.19: Stream out the GDS file (gds.py).

11.6.15 Design Rule Checks (DRC)

Execute Listing 11.20 for the design rules check.

11.6.16 SPICE Extraction

Execute Listing 11.21 for the SPICE extraction.

11.6.17 Layout vs. Schematic (LVS)

Execute Listing 11.22 for the layout vs. schematic check.

11.7 Tiny Tapeout

Tiny Tapeout is a project that takes the idea of a multi-project wafer to the next level. Tiny Tapeout uses one tile of an MPW shuttle on Skywater130 or IHP and divides that tile into 512 smaller tiles. Therefore, the production cost can be further split, allowing for the sale of one tile for less than 100 USD.

Tiny Tapeout uses GitHub actions to harden user designs. Therefore, this is the easiest way to produce a chip. No tools need to be installed locally. The project is configured within two files: `config.json` and `info.yaml`.

11.7.1 Local Hardening

It is also possible to run the hardening locally, avoiding the long latency of GitHub actions. Follow the instructions on the TT website.

However, we can also do the local hardening with our installation of LibreLane instead of using the Docker image. After installing the Python dependencies into a Python virtual environment, execute the Python script to generate the configuration:

```
./tt/tt_tool.py --create-user-config
```

```
DRC = Step.factory.get("Magic.DRC")

drc = DRC(state_in=gds.state_out)
drc.start()
```

Listing 11.20: Design rule checks (drc.py).

```
SpiceExtraction = Step.factory.get("Magic.SpiceExtraction")

spx = SpiceExtraction(state_in=drc.state_out)
spx.start()
```

Listing 11.21: Spice extraction (spice.py).

```
LVS = Step.factory.get("Netgen.LVS")

lvs = LVS(state_in=spx.state_out)
lvs.start()
```

Listing 11.22: Spice extraction (lvs.py).

This command generates the file usr_config.json in folder src with information extracted from the info.yaml file. This file is merged with the config.json into config_merged.json. We can now harden the design by running:

```
librelane config_merged.json
```

11.8 wafer.space

wafer.space is a new MPW service started by Tim Ansell and Leo Moser. It uses GF130 and offers 1000 dies with a user space of 20 mm^2. The project template is just the pad ring and a counter as an example design.

GLOSSARY

SoC
: System-on-Chip. An integrated circuit that integrates all major components of a computing system onto a single die.

ASIC
: Application Specific Integrated Circuit. An integrated circuit designed and fabricated for a single dedicated purpose, as opposed to general-purpose computers.

GPIO
: General-Purpose Input/Output. A configurable analog or digital signal pin on an integrated circuit.

JTAG
: Joint Test Action Group. An industry-standard interface (IEEE 1149.1) used for boundary-scan testing, in-circuit debugging, and programming of integrated circuits.

UART
: Universal Asynchronous Receiver-Transmitter. An asynchronous, half-duplex serial communication bus using two signals (RX, TX).

SPI
: Serial Peripheral Interface. A synchronous, full-duplex serial communication bus between a controller and a peripheral using four signals: clock (SCLK), chip-select (CS), controller-out peripheral-in (COPI), and controller-in peripheral-out (CIPO).

ICN
: Interconnect Network. The on-chip fabric that routes data transactions between initiators (e.g., CPUs) and targets (e.g., memories, peripherals) within a SoC.

OBI
: Open Bus Interface. A lightweight, open-standard on-chip bus protocol designed for connecting processors and peripherals inside a SoC.

APB
: Advanced Peripheral Bus. A low-bandwidth, low-power bus from the ARM AMBA family, intended for connecting slow peripherals such as timers, UARTs, and GPIO controllers.

AXI
: Advanced eXtensible Interface. A high-performance, high-frequency on-chip bus protocol from the ARM AMBA family that is suitable for memory controllers and high-bandwidth IP blocks.

XML
: eXtensible Markup Language. A human-readable, hierarchical text format used to encode structured data.

HAL
: Hardware Abstraction Layer. A software layer that provides a high level interface to hardware peripherals, allowing firmware and application code to be written without direct dependence on register addresses or hardware-specific details.

API
: Application Programming Interface. A defined set of functions, procedures, or protocols that allows one piece of software to interact with another.

FPGA
: Field-Programmable Gate Array. A reconfigurable integrated circuit containing an array of programmable logic blocks and interconnects that can be configured after manufacturing to implement arbitrary digital circuits.

CPU
: Central Processing Unit. The primary processor in a computing system, responsible for fetching, decoding, and executing instructions.

RTL
: Register Transfer Level. An abstraction level used to describe digital circuits in terms of the flow of data between registers and the logic operations performed on that data. RTL descriptions are typically written in a hardware description language.

HDL
: A specialized language used to describe the structure and behavior of electronic circuits, enabling simulation and synthesis of digital logic into physical hardware.

CTS
: Clock Tree Synthesis. A physical design step that inserts clock buffers and builds a hierarchical distribution network to deliver the clock signal to all sequential elements with controlled insertion delay and minimised skew.

STA
: Static Timing Analysis. A method of verifying the timing correctness of a digital circuit without simulation. STA exhaustively checks all paths in the design for setup and hold violations across all specified operating conditions (PVT corners).

PVT
: Process, Voltage, Temperature. The three principal sources of variation that affect the speed and power of an integrated circuit. Synthesis and sign-off must be performed across multiple PVT corners to ensure the design meets timing under all supported operating conditions.

DRC
: Design Rule Check. An automated verification step that checks a physical layout against the foundry's manufacturing rules (minimum widths, spacings, enclosures, density constraints, etc.) to confirm that the design can be reliably manufactured.

LVS
: Layout Versus Schematic. A physical verification step that extracts the netlist implied by the layout geometry and compares it against the reference schematic or gate-level netlist to confirm that the two are electrically equivalent.

GLS
: Gate-Level Simulation. Simulation performed on the gate-level netlist produced by synthesis, optionally back-annotated with timing information from a Standard Delay Format (SDF) file. GLS verifies that the synthesised circuit is functionally and temporally correct.

LEF
: Library Exchange Format. A file format that describes the physical abstraction of standard cells and macros (cell outline, pin locations, and obstruction layers) for use by place-and-route tools.

SDC
: Synopsys Design Constraints. A standard Tcl-based format, originally developped by Synopsys, used to specify timing constraints for a design.

SDF

Standard Delay Format. A file format that stores cell and interconnect delay values extracted from a routed layout.

BIBLIOGRAPHY

[ACFogacca+19] Tutu Ajayi, Vidya A. Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B. Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, Geraldo Pradipta, Sherief Reda, Mehdi Saligane, Sachin S. Sapatnekar, Carl Sechen, Mohamed Shalan, William Swartz, Lutong Wang, Zhehong Wang, Mingyu Woo, and Bangqi Xu. Invited: toward an open-source digital flow: first learnings from the openroad project. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 76. ACM, 2019.

[BVR+12] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In *The 49th Annual Design Automation Conference (DAC 2012)*, 1216–1225. San Francisco, CA, USA, June 2012. ACM.

[GS20] Ahmed Ghazy and Mohamed Shalan. Openlane: the open-source digital asic implementation flow. In *Workshop on Open-Source EDA Technology (WOSET)*. 2020.

[IEE20] IEEE. Ieee standard for universal verification methodology language reference manual. *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*, ():1–458, 2020. doi:10.1109/IEEESTD.2020.9195920.

[SV03] A. Sangiovanni-Vincentelli. The tides of EDA. *IEEE Design & Test of Computers*, 20(6):59–75, 2003. doi:10.1109/MDT.2003.1246165.

[Sch19] Martin Schoeberl. *Digital Design with Chisel*. Kindle Direct Publishing, 2019. available at https://github.com/schoeberl/chisel-book.

[Sch25] Martin Schoeberl. Introduction to chip design using open-source tools. https://www.imm.dtu.dk/Âămasca/chip-design-book.html, 2025.

[SDP+25] Martin Schoeberl, Hans Jakob Damsgaard, Luca Pezzarossa, Oliver Keszocze, Erling Rennemo Jellum, and Scott Beamer. Scala defined hardware generators for chisel. *Microprocessors and Microsystems*, 117:105182, 2025. doi:https://doi.org/10.1016/j.micpro.2025.105182.

[SE20] Mohamed Shalan and Tim Edwards. Building openlane: a 130nm openroad-based tapeout-proven flow : invited paper. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 1–6. 2020.