ASIC Design von Hardwarebeschleunigern für RISC-V

ASIC Design of Hardware-Accelerators for RISC-V

Lecturer	Patrick Karl
	patrick.karl@tum.de
Tutor	Florian Gruber
	florian2000.gruber@tum.de
Affiliation	Technical University of Munich, Germany
	TUM School of Computation, Information and Technology
	Chair of Security in Information Technology

Contents

1 Scope of the Lab

1.1 What the Lab is about

This lab aims to introduce the basic workflow of designing an actual chip and should give an insight into commercial tools commonly used in practice. In order to reach that goal, the lab consists of two major parts.

The first part focuses on the frontend design and includes the following aspects:

- **Familiarize** with a given RTL design. This includes understanding the basic architecture and understand how a compiled source code actually interacts with the processor's underlying hardware. For that, a given source code can be evaluated and simulated.
- **Adjust** the given RTL design. This includes designing a hardware accelerator for cryptographic purposes and integrating it into the given processor system. Finally, the given source code shall be modified such that the computation is offloaded to the hardware accelerator.
- **Evaluate** the impact of the hardware acceleration. This includes measuring the performance impact, identifying the remaining performance bottlenecks but also investigating the effect on memory consumption.

After extending the given platform with a dedicated hardware accelerator, the second part of the lab focuses on the backend flow. This part actually transitions from RTL code to the physical aspects of designing a chip:

- **Logical Synthesis:** After changing the RTL design, the given synthesis script must be adjusted accordingly. In addition to that, preliminary reports on timing and area cost can be evaluated given the generated netlist.
- **Physical Synthesis:** With the previously generated netlist, the physical synthesis can be conducted. This includes for instance the creation of a proper floorplan, power routing and clock tree synthesis.

- **DRC Check:** The final step includes solving potential DRC violations and streaming out the GDSII data which then includes all the information the fab requires to manifacture the chip.

To guide you through the above mentioned steps, there will be information about the development environment and design platform on the following pages. In addition to that, several code snippets, scripts and templates for both the frontend and the backend are already provided and serve as baseline for the whole design flow.

1.2 What the lab is not about

Although there will be several hints and explanations on the following pages, the lab does *not* serve as a step-by-step explanation throughout the design process. That means students are required to work *with* the provided setup, i.e. prove that they are capable of diving into a given setups, understand dependencies, and analyze and adapt several scripts using the provided documentation.

1.3 Prerequisites

To successfully conduct the lab, it is beneficial to have previous knowledge in the domain of RTL design using hardware description languages like VHDL, Verilog or SystemVerilog, as well as previous knowledge of simulation tools like Modelsim, Xcelium or Xilinx Vivado. Furthermore, having a solid understanding of digital design basics is crucial for being able to follow the lab content. Besides the design aspects, it is beneficial to have a basic understanding of cryptology and cryptographic operations as the first part of the lab focuses on accelerating cryptographic operations.

2 Lab Environment

For the lab, each student is given a separate Gitlab repository which contains three directories:

- **doc** contains all the documentation given for conducting the lab, i.e. this tutorial documentation as well as the documentation for the PULPino platform.
- **pulpino_clean** basically contains the entire platform that is used throughout the lab. This includes all the RTL code, simulation environment, Makefiles for software compilation as well as the actual software that will be running on the implemented RISC-V processor.
- **reports** is an empty directory where each student is asked to save generated reports to. What kind of reports are requested will be explained in the following sections.

2.1 pulpino_clean

As previously mentioned, pulpino_clean is the main directory containing the RTL design as well as software that will be running on the RISC-V processor. In addition to that, it provides some basic scripts for the backend flow described later. The structure of pulpino_clean is depicted in Fig. 2.1 and contain the following files:

- **COMPILE** contains the C code that will later run on the RISC-V microcontroller as well as corresponding Makefiles and linker scripts. It is divided into several subdirectories providing some utility functions and drivers for the RISC-V core. In this lab, you are only asked to make your changes under `src`. This is where you can find the code for the benchmark application.
- **GENUS** is a Cadence tool used for logical synthesis, i.e. for generating a netlist out of a HDL description. The directory contains several scripts to conduct the synthesis step.
- **INNOVUS** is the Cadence tool used in the backend flow to create a floorplan and route

```
pulpino_clean
├── COMPILE
│   ├── compile
│   ├── lib
│   ├── src
│   └── string-lib
├── GENUS
│   ├── constraints
│   ├── outputs
│   ├── reports
│   └── scripts
├── INNOVUS
│   ├── constraints
│   ├── outputs
│   ├── reports
│   └── scripts
├── RTL
│   ├── ips
│   ├── rtl
│   ├── rtl_synthesis
│   └── tb
└── XCELIUM
    └── scripts
```

Figure 2.1: Directory structure

everything. The directory contains several scripts for the different steps required in the backend.

- **RTL** is the directory containing all the HDL code implementing the RISC-V core and corresponding microcontroller. It further provides some testbenches used during simulation. Within this lab course, you will have to make several adjustments in order to integrate the accelerator into the provided system.

- **XCELIUM** is Cadence's simulation tool that will be used throughout the lab. There are already some scripts provided for simulation.

Within some of the directories, there are further subdirectories which we do not list in detail here. Especially within the subdirs in `RTL`, there are many more subdirs for the different peripherals and modules of the microcontroller. How the system looks like is subject to the next section.

2.2 PULPino microcontroller

Fig. 2.2 provides an overview of the PULPino microcontroller[1]. It is based on a 4-stage pipelined RISC-V core called RI5CY (shown in Fig. 2.3), that implements the RV32IM[F]C instruction set. That means that it implements the 32-bit base integer set (I), multiplication extensions (M), optional single-precision floating point instructions (f) as well as the compressed instruction set (C). In Fig. 2.2, you see that there are severall peripherals (I2C, UART, SPI etc.) attached to the system bus. Within this lab, your first goal will be to design a hardware accelerator that will be connected to the system just as one additional peripheral. As the memory is implemented as SRAM, program code has to be written into the memory

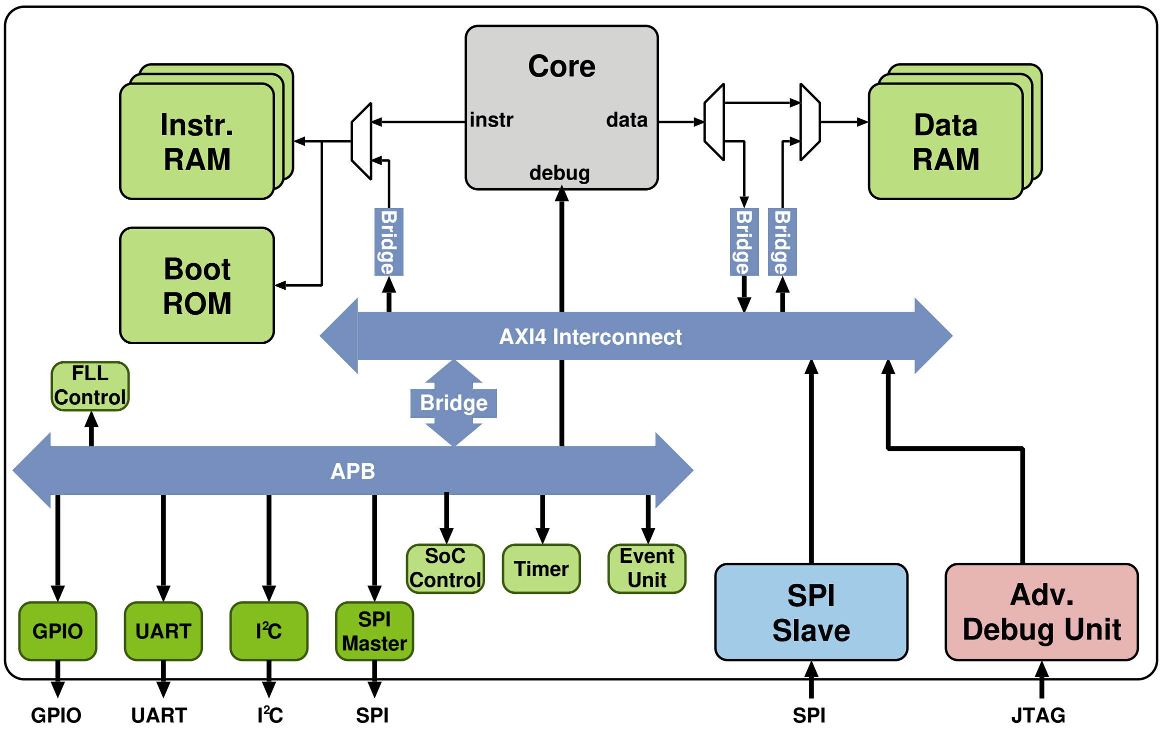

Figure 2.2: PULPino system overview [2]

each time the system is powered (or simulated). This is done via the SPI interface. When simulating code, you will therefore see some activity on the SPI interface right at the beginning. Afterwards, there is a `fetch_enable` signal that triggers the code execution when going from low to high.

[1]https://github.com/pulp-platform/pulpino

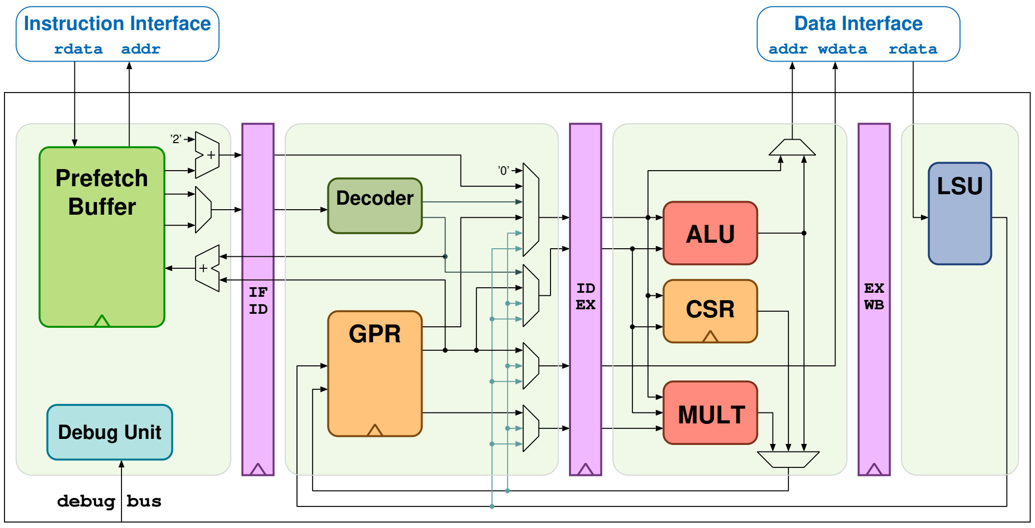

Figure 2.3: RISC-V core RI5CY [2]

2.3 PCs and Tooling

In order to conduct the lab you can use the computers in the lab-room **2947** – you can login with your usual lrzid and password. As some of the tools – especially in the backend – require quite some computational time, it is recommended to use `tmux` or `screen` sessions, such that one can detach while waiting for the computation to end. If you are not familiar with `tmux` or `screen` yet, I would recommend reading a quick tutorial as for example https://www.hamvocke.com/blog/a-quick-and-easy-guide-to-tmux/. Furthermore, to use the commercial Cadence tools, the script `setup_cadence.sh` must be sourced via `. setup_cadence.sh` in every new terminal session. Please note that in order to be allowed to use these tools, you must sign the corresponding NDA. In the remainder of the documentation, all paths are assumed to be relative to `pulpino_clean`.

Compiling source code: To compile the C code, a suitable RISC-V compiler must be available. For the lab, we provide a precompiled version that can be loaded from the lab computers. To load the compiler, execute the following every time you open a new terminal session:

```
1. module use /nas/ei/share/sec/tools/modulefiles
2. module load risqv-ht/toolchain/aquorypt-2021
```

Then, you can compile the provided C code as follows:

```
1. cd COMPILE/compile
2. make clean && make keccak_bench
```

This generates several `.bin` and `.elf` files and converts them to a simple textfile `spi_stim.txt` that contains the memory addresses and corresponding data words for the instruction and data memory of the RISC-V core.

Simulating Code: For RTL simulation the Cadence Xcelium simulator will be used. The corresponding scripts can be found in `XCELIUM/scripts/`. To simulate a C program on the RISC-V core, make sure that the code was compiled as explained previously and proceed as follows:

```
1. cd XCELIUM/work
2. ../scripts/run.sh
```

This should invoke the simulator and start simulating your code. Note, that you might have to create the directory `work` first via `mkdir work`. By default, the simulation is started in batch mode without graphical interface. When integrating and inspecting your accelerator, it might be desireable to look at specific waveforms. To start the simulator in graphical mode, attach the `-gui` flag to the `xrun` command in `XCELIUM/scripts/run.sh` such that it becomes

```
xrun $1 -64bit -f ../scripts/sim.args -access +rwc \
    -define STIM_PATH=\"${STIM}\" \
    -define PULP_FPGA_EMUL \
    -disable_sem2009 -v93 -timescale 1ps/1ps -simvisargs \
    '-input ../scripts/sim_wave.tcl' -top tb -gui
```

Modifying the RTL design: After you designed your accelerator, it has to be included into several scripts to simulate and synthesize it. For simulation, the corresponding Xcelium scripts must be adapted:

```
1. cd XCELIUM/scripts
2. vim sim.args
```

Note, that the variable `${RTL}` corresponds to the directory `RTL`, whereas `${IPS}` corresponds to `RTL/ips`. In addition to simulation, the corresponding synthesis script must be adapted:

```
1. vim src_files.tcl
```

For simplicity, you can simply add your files to the `SOURCES` list.

Synthesize RTL to Netlist: In order to generate a netlist of the design, Cadence Genus is used. All the scripts are already provided such that you only have to invoke the the tool and source the script:

```
1. cd GENUS/work
2. genus
3. source ../scripts/synthesis.tcl
```

The command in line 2 starts Genus whereas line 3 sources the corresponding synthesis script. Have a look at the script and make yourself familiar with the commands by reading them up in the documentation. Note, that if you want to add new modules/files to the original RISC-V design, you have to register the files in the scripts as descriped previously.

Backend Flow: The process of converting a netlist into a fully described chip design is called backend flow. In this lab we use Cadence Innovus for that step. There are already a lot of scripts provided for the backend. In order to start the process, invoke the tool and source the corresponding file:

```
1. cd INNOVUS/work
2. innovus
3. source ../scripts/phy_syn.tcl
```

When Innovus is invoked, a graphical window opens showing the current state of the design. In addition to that, the command line stays active which allows to enter addtional commands. It is also recommended that you open the `.tcl` scripts in a separate editor and copy the commands to the command line. With this procedure you can immediately see the effects of the commands. Alternatively you can use the `suspend` and `resume` commands to pause and continue the process. Note, however, that you have to insert the `suspend` commands before sourcing the script in line 3.

Documentation: The following documentations are recommended as introduction to the topic and to look up the tcl commands supported by the tools. It should be used to get a better understanding on the commands used especially in the backend flow. Please note that

in terms of the NDA you are not allowed to download the documents to your own computer.
In the following, the variable `$DOC` stores the path `/nas/ei/share/tools/cadence`.

- **PULPino:**
  `https://pulp-platform.org/docs/pulpino_datasheet.pdf`

- **Keccak:**
  `http://dx.doi.org/10.6028/NIST.FIPS.202`
  `https://keccak.team/`

- **Xcelium:**
  `$DOC/XCELIUM/XCELIUM_21.03.009/doc/xrun/xrunTOC.html`

- **Genus:**
  `$DOC/GENUS/GENUS_19.14.000-ISR4/doc/genus_user/genus_userTOC.html`
  `$DOC/GENUS/GENUS_19.14.000-ISR4/doc/genus_comref/genus_comrefTOC.html`

- **Innovus:**
  `$DOC/INNOVUS/INNOVUS_20.11.000-ISR1/doc/UGcom/UGcomTOC.html`
  `$DOC/INNOVUS/INNOVUS_20.11.000-ISR1/doc/TCRcom/TCRcomTOC.html`
  `$DOC/INNOVUS/INNOVUS_20.11.000-ISR1/doc/innovusTCR/innovusTCRTOC.html`

3 Part I: Hardware Accelerator Design

3.1 Goal of Part I

The goal of the first part is the design and integration of a hardware accelerator for cryptographic operations. More specifically, a **Keccak accelerator** for the hash standard **SHA-3** [1] should be developed and integrated that implements the `shake128` function of the standard. To do so, you don't have to start from scratch – there is an accelerator template provided for easier integration into the system. For the actual Keccak implementation, you can take an open-source implementation from the official Keccak team. Both the accelerator template and some information regarding the Keccak implementation is given in the following sections.

3.2 Accelerator Template

3.2.1 Design Overview

Fig. 3.1 shows the architecture of the accelerator template. The design consists of two major entities, the `accelerator-top-wrapper` and the `accelerator-wrapper`:

accelerator-top-wrapper: The top-level features an `AXI4-slave` port for connecting the module to the central bus. The AXI4 memory request is translated to a low-level memory bus by the `AXI4-to-Mem` module. Via this low-level memory bus, the AXI4 master can access two different memory domains, i.e. the FF-based `configuration register` and the accelerator's `data memory` (RAM). The infrastructure of the `accelerator-top-wrapper` is already given and should not be modified much, because it handles the communication between accelerator and the RISC-V core. The only exception to that is the parameterization of the `configuration register` and it's connections with the control and status vector.

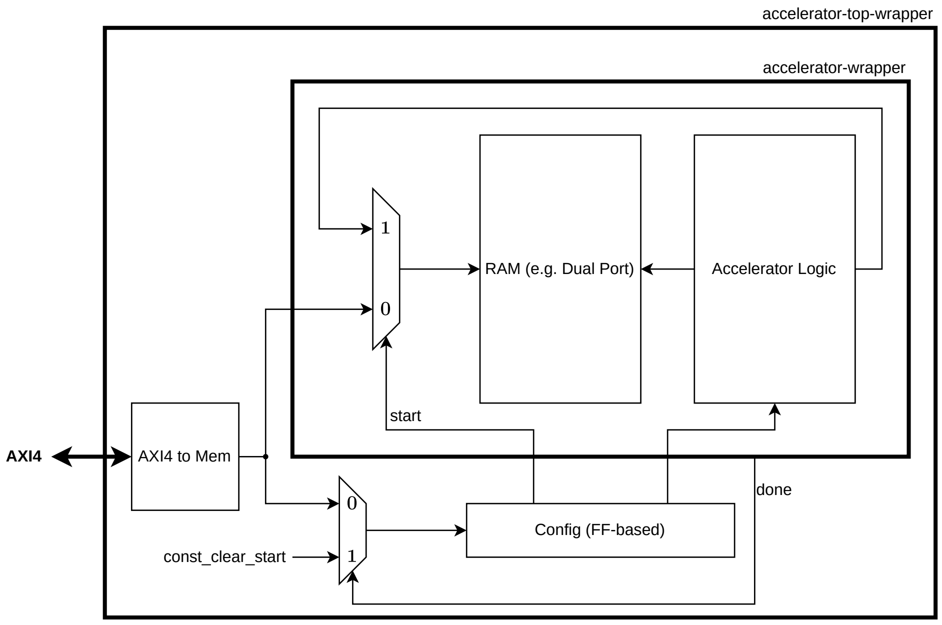

Figure 3.1: Architecture of the accelerator template

accelerator-wrapper: The wrapper features a low-level memory bus slave port and some user-defined ports for the configuration of the accelerator that comes from the `configuration register`. In addition to that, there are a few mandatory control ports (e.g. start, done). The content of this wrapper is under the designer's responsibility. Here the accelerator functionality should be implemented. The designer can chose any arbitrary memory type, how it is accessed etc. However, one has to make sure that the memory can be accessed by the low-level memory bus interface when the accelerator is in idle mode, such that the RISC-V core can write/read the data memory.

config register: The configuration register can be accessed via the low-level memory bus coming from the `AXI4-to-Mem` module. It stores two types of data i.e. `control` data and `status` data.

The `control` data is input to the `accelerator-wrapper` and can be used to configure the module at runtime, for instance switch between different modes of operation. The data is sent to the module in RAM style, i.e. it is a two dimensional array where the first dimension would be the address, the second dimension the data word. As a result, all the control data is accessible in parallel. The number of words as well as the word width can be configured (word width in multiple of bytes). Fig. 3.2 shows an example of 4 control words with 32-bit each (labeled with CTRL). Note, that the lowest significant byte of the first control word entry is reserved, as bit 0 is used as start bit. Therefore, the number of control words is required to be > 0. As shown in Fig. 3.1, a 'done' pulse coming from the accelerator is used to clear the start bit of the control vector, such that the accelerator will not start processing the same data again after finishing a computation. All control information can be written and read out from the RISC-V core.

The `status` information is an output from `accelerator-wrapper` point of view and can be used to store status information of the module, e.g. current computation state, error messages or any kind of data that would be useful to monitor at runtime. Fig. 3.2 shows an example of 4 status words (labeled with STAT). Again, one notes that the lowest significant byte is reserved. It encodes some module status and error information. More precisely, bits 0-3 are used to distinguish between IDLE and RUNNING state, bits 4-7 encode some error message like ERR_OKAY or INVALID_CONFIG. For more information, have a look at the example design and at `accel_cfg.sv`, where these messages are encoded. Due to that reserved byte, it is mandatory to have the number of status words configured for the config register to be > 0. All status information is readable by the RISC-V core.

3.2.2 Memory Arrangement

The system-wide AXI4 bus is configured to have an address width of 32-bit. As some address space is used for other domains and one has to distinguish between different AXI4 slaves, every accelerator can make use of 20 bits of the address.

For example, let our accelerator be mapped to address `0x1E10_0000`. Then the accelerator can freely use the lowest 20 bits marked in blue. Inside the accelerator, we have now two memories we want to access, i.e. the `configuration register` and the `data RAM`. Therefore, we use the MSB of the internal address to distinguish between both domains. The configuration register is assigned with address space `0x0_0000` to `0x7_FFFF`, whereas the data memory can use address `0x8_0000` to `0xF_FFFF`. Dividing the memory space by 2 is not very efficient in terms of memory usage, because we most likely only need a few words for the `configuration register`. However, it makes it very easy to distinguish between data and config domain and in addition to that, having a 19-bit address for data memory should still be sufficient for most usecases.

The address space for the configuration address is again divided into control and status domain. For that, the number of bits required to access all the control and status words is determined. The register expects the address to be word aligned. Thus, for a data width of 32-bit, the lowest 2 bits would always be zero.

Fig. 3.2 depicts an exemplary address map for an accelerator using 4 control words and 4 status words. Because memory addresses are expected to be word aligned, the 2 least significant bits of the address are 0. Then, the next four addresses are used for the configuration, and the next four words are used for status information.

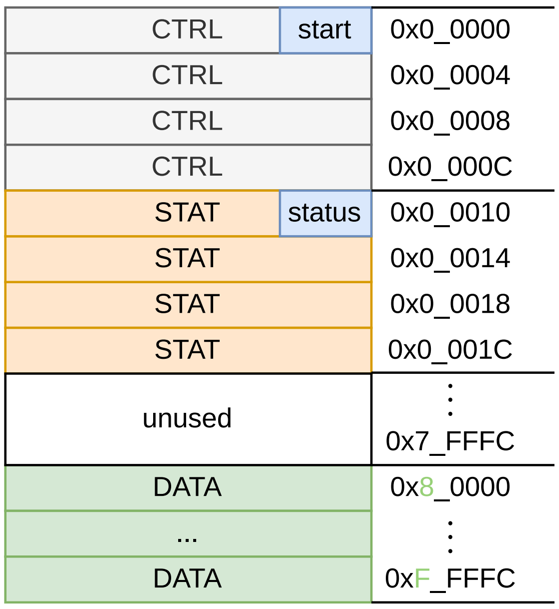

Figure 3.2: Exemplary memory mapping

3.2.3 Example Flow Chart

Fig. 3.3 shows an example how to use an accelerator. The idea is straightforward: Write the data to be processed in the memory and optionally write the control information in the `configuration register`. Afterwards, start the computation by writing `0x01` into the start byte of the first control word (shown in Fig. 3.2). Wait until the accelerator finished computation - this will most likeley be indicated by an interrupt or by simply polling the start bit again. Finally, read back the result from the data memory.

3.2.4 Example Design

The accelerator template features an example design with a simple adder and corresponding testbench. For the testbench, the `AXI4-to-mem` module must be bypassed. Thus, it is required to comment the `AXI4-slave` port of the module and use the low level bus as interface. In addition to that, the `AXI4-to-mem` module must be commented as well as the locally defined signals.

The example adder reads all data from data memory, adds +1 to every byte and writes the data back to memory. For this computation, both ports of the dual port RAM are used. The number of data words the adder reads from memory is configured in the 2nd byte of the first

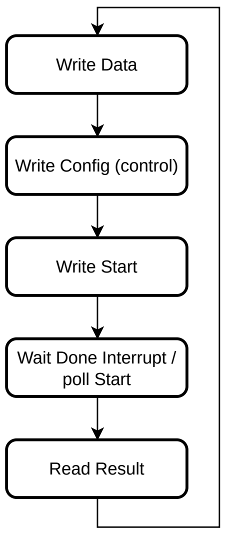

Figure 3.3: Exemplary flow chart

control word. You can have a look at lines 156-166 `accel_top_wrapper.sv` to get an idea how you can connect the control / status vectors to your design.

In the testbench, the following actions are performed sequentially:

1. The data that is to be processed is written into the data memory
2. The number of data words (in data memory) as well as the start bit is written in the control register
3. While the adder computes the result, the testbench reads the control and status values, just to show the functionality
4. After control and status is read, the testbench waits for done interrupt
5. Finally, the processed data result is read back from data memory

There are a few things that can be parameterized in the design, for instance the data width and the number of control / status words. You can just play around a little bit to get an idea how everything works.

3.3 Keccak Implementation

As mentioned before, you do not have to start from scratch implementing the Keccak permutation. Furthermore, you can use the provided implementations that can be found on the Keccak-team website https://keccak.team/. Under https://keccak.team/hardware.html you find three different VHDL implementations: a high-speed core, a compact co-processor and a mid-range implementation. You can take the implementation of your choice depending on your justification. Fig. 3.4 shows the sponge construction used in the Keccak scheme.

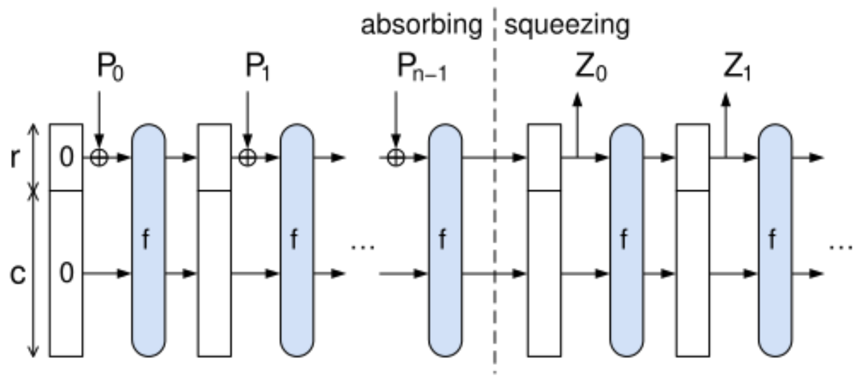

Figure 3.4: Sponge construction used in Keccak

It is recommended that you take your time understanding the SHA-3 standard and inspect the corresponding implementations before choosing one core.

3.4 Tasks and Hints

1. Make yourself familiar with the software environment and simulate the given C code for `shake128`. For that:
   - Have a look at the C code under `COMPILE/src` and the Makefile at `COMPILE/compile`
   - Go to `COMPILE/compile` and type `make all` to compile the code
   - From `XCELIUM/work` type `../scripts/run.sh` to start the simulation
2. Play around with the `shake128` C code and measure the performance as well as the stack consumption for input sizes of 4B, 8B, 16B, 32B and output sizes of 16B, 32B, 64B.

3. Design the hardware accelerator using an open-source Keccak implementation from the Keccak team.
    - Use the accelerator template and think about what values you need to configure, e.g. things like input/output sizes
    - Use the provided dual-port RAM for storing the input and output for the Keccak core
    - Design a finite state machine (FSM) that controls the accelerator, i.e. starts reading the input from memory after being triggered by the `start` signal, setting the Keccak core's control signals and writing back the output of the Keccak core into the memory. To reach the goal, inspect the provided example adder first and think about how you would design your FSM.
4. Integrate the accelerator into the system.
    - Add all your source files to `./src_files.tcl` (for synthesis with Cadence Genus) as well as `XCELIUM/scripts/sim.args` (for simulation with Xcelium)
    - Instantiate your accelerator top level in `RTL/rtl/pulpino_top.sv`. In the same file, extend the signal `slaves` in line 122 by one additional slave. Next, inspect PULPino's datasheet to find a suitable address range for your accelerator. Finally, connect your accelerators slave port with the additional slave signal you previously defined and define it's address range in the `axi_interconnect_i` instance in lines 296-316.
5. Adjust the C code in `COMPILE/src/bench_targets/main.c` such that it uses your accelerator instead of the software based `shake128` function and simulate your design to verify it's functionality.
    - You can use simple pointer (de-) referencing to write to the specific memory addresses. For instance `uint32_t *acc = (uint32_t *) 0x12345678;` would define a pointer `acc` that points to address `0x12345678`
    - With that, you can use your accelerator by writing the input data to the corresponding addresses, configure it and trigger it's computation. Afterwards, you can read the result from the specific addresses.
6. As before, measure the performance as well as the stack consumption for input sizes of 4B, 8B, 16B, 32B and output sizes of 16B, 32B, 64B using your implementation and compare it with the software-based version of `shake128`.

4 Part II: Physical Design – From RTL to GDSII

In the second part of the lab, we want to design a chip of our modified RISC-V design using a 130nm Skywater technology. In order to fabricate an ASIC, a fab requires some sort of data format that contains all the geometries for the different layers of our chip, the so called GDSII format. In the following, a short introduction on this process is given. With that, the goal of the lab is to adjust the given workflow to the extended design.

4.1 Netlist Generation

In the first step, our new RTL description that is purely written in HDL languages like VHDL, Verilog or SystemVerilog must be transferred into a netlist. A netlist then only contains the basic modules of the design mapped to cells or macros from our standard library. In this lab, we work with Cadence Genus as synthesis tool. Therefore, all the required scripts are found under `GENUS/scripts` – in fact, we only require a single script `synthesis.tcl`.

In the first couple of lines, some path variables and generic setting are defined. This is not really specific to a design but is rather used to make the code and output more readable. Let's go briefly over the remaining parts of the script.

- **LIBRARY PATHS:** Another script called `library_paths.tcl` is sourced. This script basically contains a lot of paths to different files of our standard library. Examples for that are `.lef` and `.lib` files that are required during the backend flow.
- **READ LIBS + LEFS:** The above described `.lef` and `.lib` files are read. A Library Exchange Format (`.lef`) file contains physical information of the layout and Design Rule Checks (DRCs) of a cell. It can be seen as a kind of abstraction of the cell's layout. A Liberty file (`.lib`) contains timing and power parameters associated with the different cells.
- **READ RTL:** In this part, the HDL code i.e. our (System-) Verilog or VHDL code is loaded.

- **ELABORATE:** Here the design is elaborated. That means, the design is compiled and checked for syntax issues. Furthermore, all module instances are actually represented and generics/parameters/constants etc. are evaluated and propagated. Finally, specified constraints like number of pins, timing information, clock frequency etc. are read.

- **SYNTHESIS:** Once elaboration is successfull, the design is synthesized.

- **WRITE RESULTS:** Reports of the synthesized netlist are generated and the artifacts of the synthesis step are saved.

4.2 Floorplanning, Cell Placement, Power Routing, Clock Tree Synthesis, Routing

The goal of the remaining backend flow is to take the generated netlist and turn it into a physical design. On an abstract level this includes floorplanning, cell placement, power routing, clock tree synthesis and final routing. If no design rule violations (DRCs) occur, the final design can be sent to the fab. Please note that we won't aim to solve DRCs during this lab, as this will most likely be too time consuming and is out of the lab's scope.

In this lab, the steps described above are performed by Cadence Innovus. The corresponding scripts can be found at `INNOVUS/scripts`, whereas the main script is called `phys_syn.tcl`. Let's have a look at that script:

- **LOAD DESIGN:** Some settings are defined and another script called `load_design.tcl` is sourced. This script further sources `library_paths.tcl` to set the paths to the standard cells again as well as `Default.globals` that contains some initial configurations. Afterwards, some filler cells are defined and the netlist is checked. Finally, the floorplan is created via sourcing `floorplanning.tcl`

The following sections describe the steps performed in `floorplanning.tcl`

- **FLOORPLANNING:** The floorplanning is most likely the most time consuming task of a backend designer. First of all, several variables are defined that make the following code more readable. At *Specify Floorplan*, the floorplan is specified, i.e. the dimensions of the core area. In addition to that, the memories of the RISC-V core are placed and routing Halos are put around them.

- **GLOBAL POWER NETS:** These commands are used to correlate the power/ground pins of the cells and the memories with the global power/ground pins. Then, a power/ground ring around the core is created as well as some vertical and horizontal power/ground stripes that connect the cells and memories with the external power.

- **SPECIAL ROUTE:** In this section, all the power/ground pins of the standard cells and macros are actually connected and routed such that they are connected to the power grid.

After that, we can go back to the `phys_syn.tcl` script:

- **CELL PLACEMENT:** Now that 1) the floorplan is defined, 2) macros are set to fixed positions and 3) the power grid is defined, the next step is to place all the standard cells in the design. At this stage, several different special cells are included such as *well taps*, *end caps* or *tie high* and *tie low* cells. After that step, all the cells in the design are placed to some fix positions.

- **CLOCK TREE SYNTHESIS:** After placing the cells at their corresponding positions, the clock tree is synthesized. That means, that the global clock signal is connected to all the registers and memory macros in the design. To do that, several options such at preferred routing layers are specified and some post-synthesis optimizations are performed.

- **ROUTING:** After routing the clock, the data paths between all the cells are routed, i.e. connections between the standard cells and other macros are generated. In addition to that, the router inserts buffer cells, analyzes the timing and tries to match all specified constraints. After that, some final checks are performed to verify that everything was okay. This includes for example setup and hold time checks. In addition to that, antenna cells are inserted and connected in case antenna violations are detected.

- **POST-PROCESS, VERIFICATION, REPORTS:** In the final step, we can perform some additional post-processing steps like the manual insertion of some filler cells, verification of several DRCs and shorts. Finally, after all of that is done and the design is checked for issues, the final design can be exported into a GDSII format which can be sent to the fab. However, as our design is not free of DRCs, we will not perform the generation of the GDSII data.

4.3 Tasks and Hints

In order to adapt the backend flow to your new design, proceed as follows:

1. Make yourself familiar with the environmental constraints as well as the cells of the standard library (have a look at the `.lef` and `.lib` files).

2. Generate the netlist containing your hardware accelerator. Store both the timing and area reports under `./reports`, whereas `./` is the root directory of the Gitlab repository.

    - Extend the corresponding scripts such that Genus also reads your newly added accelerator files.

3. State the difference between `syn_gen`, `syn_map` and `syn_opt`.

4. Try to figure out the purpose of special cells like *well taps*, *end caps* and *tie cells*.

5. Add the corresponding command to place the memory added by the accelerator to a desired position.

    - Have a look at the InnovusTCR documentation and read up the commands `create_relative_floorplan` and `addHaloToBlock`. Explain the purpose of a halo in general.

6. For the system clock a stable signal with small resistance is desirable. Therefore, the clock tree is usually synthesized on the higher metal layers (wider, thicker wires $\rightarrow$ less resistance). Furthermore, the clock tree is typically divided into top, trunk and leaf tree. To configure the routing tool for specific clock tree properties, so called non-default (routing) rules (ndr). Your task is to define two ndr's in the *Clock Tree Syn* part of `phys_syn.tcl`:

    - **trunk:** Define a ndr called `CTS_2W2S` for metal layers 1 to 5 that specifies a doubled width and doubled spacing (therefore CTS_2W2S). Also generate vias for it.

    - **leaf** Define a ndr called `CTS_2W1S` for metal layers 1 to 5 that specifies a doubled width but single spacing (therefore CTS_2W1S). Also generate vias for it.

    Hints: Have a look at the documentation and read up the commands `add_ndr`, `create_route_type` and `set_ccopt_property`

7. Generate the reports for gatecount and final timing and store them under `./reports`, whereas `./` is the root directory of the Gitlab repository.

List of Figures

List of Tables

Bibliography

[1] Morris J. Dworkin. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical report, jul 2015. https://doi.org/10.6028/NIST.FIPS.202.

[2] Andreas Traber and Michael Gautschi. Pulpino: datasheet. *ETH Zurich, University of Bologna*, 36, 2017. https://www.pulp-platform.org/docs/pulpino_datasheet.pdf.