Digital System Design Practicals 2021

May 1, 2021

Contents

1 Introduction

This document describes the tasks for the course "Digital System Design Practicals" for the summer term 2021. In this course, we will study how to design digital integrated circuits from the specification of the chip to the backend design. Note, the assignment sheet does not contain all exercises yet. When providing an update of the assignment, we will upload a new version to the course website and will announce it in `#dsd` and the newsgroup.

1.1 Communication Channels

We provide the following communication channels:

DSD Email. We provide the email addresses robert.schilling@iaik.tugraz.at and pascal.nasahl@iaik.tugraz.at for personal requests. Use this email only if you have a question, which cannot be discussed publicly.

Newsgroup. The newsgroup `tu-graz.lv.vlsi-design` is used for general questions for the practicals. It is the best way to share questions and answers publicly. Course instructors are going to answer your questions here. The newsgroup is also used to announce updates for the practicals. So please read it regularly!

Discord. Discord is used for questions regarding the lecture or the practicals and also for the task interviews. You need to register an account on Discord and then join the "IAIK" server. In `#getting-started`, please subscribe for the `#dsd` channel. It helps us to recognize you if you don't pick an arbitrary username, but one related to your civil name. You can use the following invitation link:

`https://discord.gg/PP6Aqk7Sr2`

- `#dsd` is a generic channel for all DSD participants to ask questions in textual form. Besides the newsgroup, it is another place to ask questions textually.
- `#GXX` is a prefix used for audio-only channels per group. You can use these channels when solving the assignments.

1.2 Digital Design Flow

The Digital Design Flow you are using is introduced in the presentation and the video tutorial. We recommend you to install Modelsim locally on your computer, as this speeds-up the development process of your HDL design.

1.3 Submissions

At the beginning of the semester, you will receive account credentials for some GitLab instance. Using git, you can submit your deliverables in your personal git repository. To submit, pay attention to the following aspects:

- You need to *tag* your commit. The tag name format is `exX`, where `X` corresponds to the respective task. For example, the git tag for the submission of Task 1 is `ex1`.
- After tagging the commit, don't forget to push your tag!
- You can check the state of your git repository by visiting your git repository in GitLab.
- If you tagged the wrong commit, you can delete the tag and tag the correct commit.

The latest possible submission deadlines for the respective tasks are given in Table 1.1. These timestamps are hard deadlines. If you participate in any submission process, you are going to get a grade at the end of the semester.

Task	**Deadline**	**Max. points**
Task 1 (Cipher Implementation)	Tue, 2021-04-27 23:59	max. 60 points
Task 2 (Cipher Integration)	Tue, 2021-06-15 23:59	max. 40 points

Table 1.1: Task submission deadlines and maximum achievable points.

1.4 Task Interviews

At the end of the semester, there will be a task interview. The date for the interviews will be organized by us and we will inform you ahead of time. The interviews cover general questions of each task and also discuss your particular solution you submitted. For your task interview, please join your individual `#GXX` channel on Discord ten minutes before the interview. This is also the appropriate place to test your microphone. The goal of interviews is to verify you did the implementation yourself, understood the topic and collect feedback on both sides.

1.5 Grading

The maximum points per task are listed in Table 1.1. Depending on your achieved points, you will get the associated grade according to Table 1.2.

0–50	Points	→	Nicht genügend (5)
51–62	Points	→	Genügend (4)
63–75	Points	→	Befriedigend (3)
76–87	Points	→	Gut (2)
88–100	Points	→	Sehr gut (1)

Table 1.2: Points to grade mapping.

1.6 Plagiarism

We will regularly check all submissions using automated plagiarism checking tools. If we detect a case of plagiarism, all involved people (the source and all sinks) will receive the grade `U` (Ungültig/Täuschung). Cases of plagiarism are handled as soon they are detected.

To avoid getting into a situation of plagiarism follow the following rules:

- Don't share code!
- Don't tell/dictate your solution to others!
- Commit regularly to show activity!

2 Task 1: Cipher Implementation

In the first task, you are implementing an accelerator for a cryptographic primitive. Resource constrained devices, such as in the Internet-of-Things, require lightweight cryptography for efficiently encrypting data. Currently, the National Institute of Standards and Technology (NIST) is performing a competition to find suitable candidates for lightweight cryptography:

https://csrc.nist.gov/projects/lightweight-cryptography

These candidates are AEAD (authenticated encryption with associated data) ciphers.

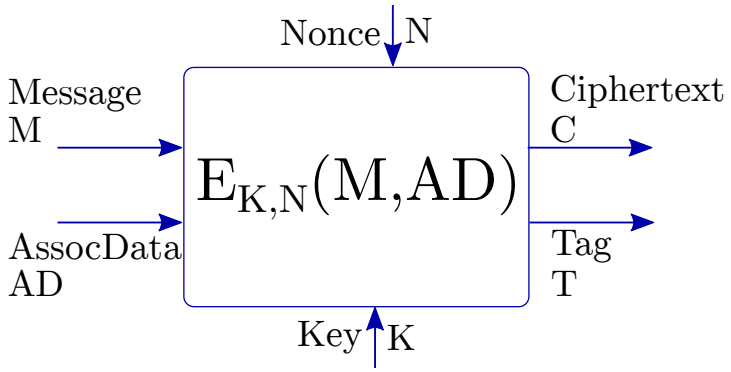

Figure 2.1: AEAD scheme.

While simple block ciphers (like a plain AES cipher) only can provide data confidentiality, AEAD ciphers can provide data confidentiality and integrity. As seen in Figure 2.1, an AEAD cipher consists of the inputs message (M), associated data (AD), nonce (N), and key (K) and of the outputs ciphertext (C) and authentication tag (T). The nonce (number used only once) ensures that a different ciphertext is produced when encrypting the same message twice even with the same key. When encrypting a message, the AEAD cipher generates the ciphertext and also the authentication tag to provide integrity. During the decryption process, the tag is used to verify the integrity of the ciphertext and an error is raised if there are manipulations on the data. Furthermore, AEAD schemes support processing of associated data. This data is not encrypted, but the computed tag also provides integrity for the associated data.

2.1 Task

The goal of this task is to implement an AEAD cipher from the NIST lightweight cryptography competition as a hardware module. Each group gets a different cipher, but with a very similar top-level interface. The specification of the cipher will contain all necessary information to implement the design.

2.1.1 Interface

Listing 1 defines the top-level interface you need to implement for the top module of your cipher. You can find the sizes `KEY_LENGTH`, `DATA_LENGTH`, and `KEY_LENGTH` in the specification of your cipher.

Listing 1 Cipher Top Module Interface.

```
module CIPHER (
  input  logic                    Clk_CI,          // Rising edge active clk.
  input  logic                    Rst_RBI,         // Active low reset.
  input  logic [KEY_LENGTH-1:0]   Key_DI,          // Encryption key.
  input  logic [DATA_LENGTH-1:0]  Nonce_DI,        // Nonce.
  input  logic [LENGTH-1:0]       DataLen_DI,      // # of data blocks.
  input  logic [LENGTH-1:0]       ADLen_DI,        // # of ad blocks.
  input  logic [DATA_LENGTH-1:0]  InData_DI,       // Plaintext and AD.
  input  logic                    InDataValid_SI,  // Master valid.
  output logic                    InDataReady_SO,  // Slave ready.
  output logic [DATA_LENGTH-1:0]  OutData_DO,      // Ciphertext.
  output logic                    OutDataValid_SO, // Slave valid.
  input  logic                    OutDataReady_SI, // Master ready.
  output logic [DATA_LENGTH-1:0]  Tag_DO,          // Tag.
  input  logic                    Start_SI,        // Start signal.
  output logic                    Busy_SO,         // Cipher busy.
  output logic                    Finish_SO        // Cipher finish.
);
```

Figure 2.2 depicts an example timing diagram of the cipher communication. This example communication performs an encryption operation and processes two blocks of associated data and two blocks of plaintext. The cipher computes two ciphertext blocks and a tag.

The initiating party applies the key, nonce, data length, and AD length and starts the cipher using the control signal `Start_SI`. Then, the cipher switches to the busy state (`Busy_SO` = 1)and processes this information. Next, the cipher processes the associated data, the input data and returns the encrypted output data. Since the cipher can encrypt more than one block during one encryption, we use a simple handshake protocol. For the input interface, the cipher waits for valid data indicated by `InDataReady_SO`=1. The initiating party applies the input data and signals this by `InDataValid_SI`=1. The cipher detects this and can process the data. The same protocol is used to transmit the encrypted data on the `OutData` interface.

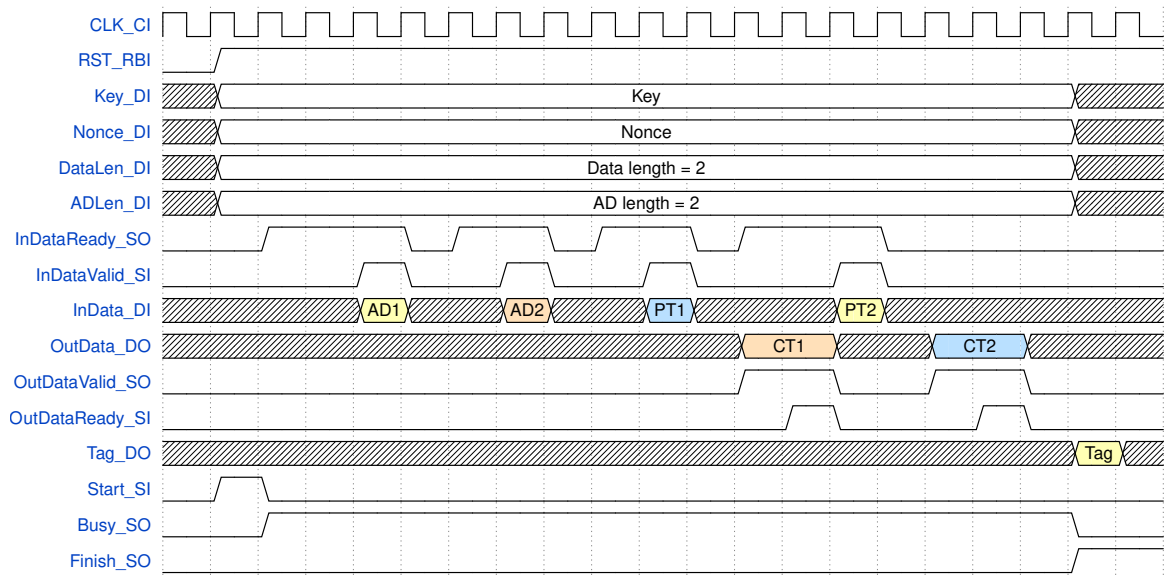

Figure 2.2: Timing diagram of the signal interface.

As indicated in the timing diagram, the cipher first processes all blocks of associated data. Then, it starts the actual encryption operation and processes each block of input data. Since the cipher cannot store the whole ciphertext, it directly transmits the computed ciphertext after its computation. Finally, after processing all data, the cipher finishes by setting the `Finish_SO` signal, releasing the `Busy_SO` signal, and applying the computed tag on the output signal `Tag_DO`.

2.1.2 Testing

The framework for the first task already provides most of the testing facility. As shown in Figure 2.3, the core of the testbench is the cipher with the top-level interface as discussed above. The testbench uses so-called drivers and monitors to hook-up the signal interface of the cipher. The drivers are responsible for applying data to the cipher. The monitors are responsible for reading data back data from the cipher and comparing it against the expected data.

To apply and read back data, both the driver and the monitor have access to the current test case, which contains all the necessary data. The test case is provided in the form of a `Stimuli` object, which is parsed from the test vector file. The test vector file (`cipher.tv`) contains a single test case per line according to the format defined in Figure 2.4. All elements are delimited with a space character. First, the test vector contains the encryption key, nonce, AD length, and data length. Next, it contains the associated data, input data, and expected output data. The number of those elements is dynamic, defined by AD length and data length. Finally, the test vector contains the expected tag.

Note, your high-level model must generate test vectors in this format. If not, you

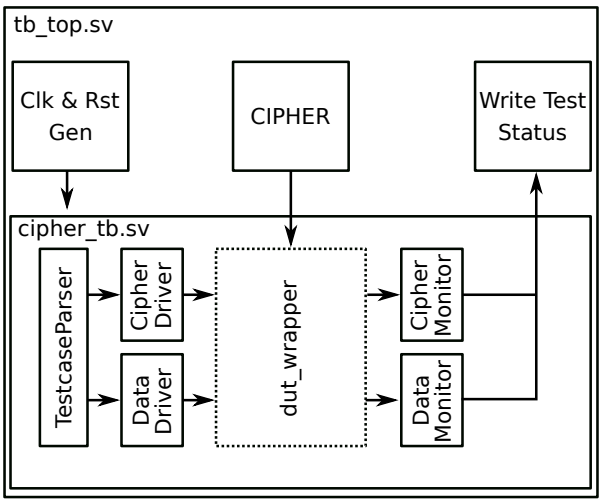

Figure 2.3: Testbench structure.

Figure 2.4: Testvector format structure.

might need to adapt the test case parser in `testcase_parser.sv`. Furthermore, your cipher might use different block lengths. Adapt the test case parser and the stimuli object accordingly.

2.2 Workflow

To complete Task 1, follow this workflow:

1. High-level modeling.
   - Write an executable high-level model of the cipher using a language of your choice (Python, C, ...).
   - Generate test vectors with this first rough model.
   - You can find a reference implementation and reference test vectors on the NIST website.
2. Architecture design.
   - Design the architecture of your cipher.
   - Define modules, interfaces, and control-logic.
3. HDL Implementation

- SystemVerilog, Verilog, VHDL

4. Test your implementation.
   - Finish the implementation of all drivers and monitors.
   - Integrate the cipher to the testbench and verify the functionality of your implementation using the generated test vectors.

Note, the HDL implementation and testing might be done in the same step.

2.3 Specification

Your implementation must fulfill the following specification:

- Design your cipher with an HDL language of your choice (SystemVerilog, Verilog, VHDL).
- Implement the AEAD version of this cipher.
- Implement one variant of the cipher (e.g. encryption, 128-bit block size).
- Use the interface described in Section 2.1.1.
- Test your cipher using the testbench framework with your generated test vectors.
- Run the synthesis on the cluster (check for timing violations, latches, ...).
- Create a design document including the following points:
  - Introduction:
    * Give an introduction into the cipher and the domain it is used.
    * Introduce the constraints of the design (power budget, throughput, security, ...).
  - Specification:
    * Highlight the functionality of the cipher.
    * Partition the ciphers's functionality into subtasks.
    * Discuss the state machines and data path of your cipher.
    * Explain the interface.
    * Extract the chip area (in GE) from the logs.
    * State the maximum frequency of the design.
  - Architecture:
    * Draw a block diagram visualizing the architecture of your design.
  - An example form is available on the course website

2.4 Grading

- Task 1: 60 points
  - Tests: 20 points
  - Design document: 5 points
  - Design: 35 points
- Deductions:
  - Coding standard: -5 points
  - Errors and warnings (e.g. latches): -5 points

2.5 Deliverables

All files must be submitted in folder `ex1` of your repository.

IMPORTANT Make sure to *commit* your files and *push* them to GitLab.

IMPORTANT Don't forget to create a tag and push the tags according to Section 1.3.

3 Task 2: Accelerator Integration

In the second exercise, you are integrating the cryptographic accelerator from the first exercise into a RISC-V System-on-Chip (SoC). You implement a cryptographic peripheral, that is memory mapped and can be accessed by the processor. Implement a small C program that programs the peripheral and performs encryption.

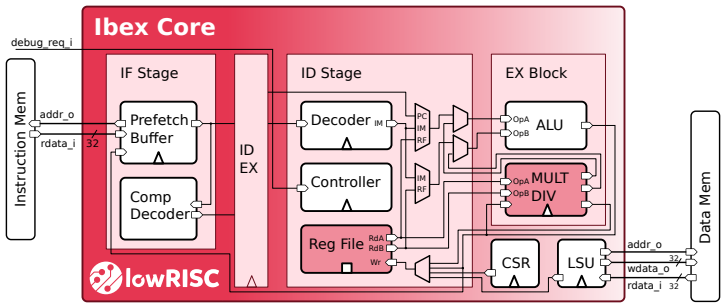

Figure 3.1: The Ibex core.

Figure 3.1 depicts the Ibex core, a 2-stage in-order 32-bit RISC-V processor, which you can find in the `ex2/src/ibex` directory of your repository.

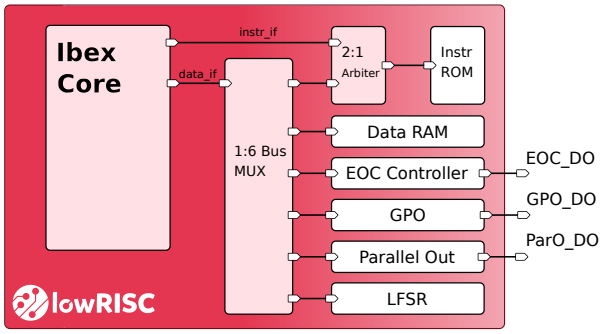

Figure 3.2: The Ibex chip.

The Ibex chip, which you can find in the top module `ex2/src/sv/ibex_top.sv` and in Figure 3.2, already includes a set of peripherals. While the parallel out peripheral allows to establish a communication between the Ibex and the testbench, the EOC controller indicates the end of the executed program. The general purpose output (GPO) peripheral allows the Ibex to set output pints of the chip directly in software. Furthermore, the Ibex chip contains an on-chip instruction ROM and a data RAM.

Interconnect. Both, the peripherals and the on-chip memory are connected to the Ibex processor using a 32-bit interconnect. While the instruction interface allows the Ibex to fetch instructions from the read-only instruction memory, the data interface is used to interact with the peripherals and the data memory. An explanation of the communication protocol can be found in the Ibex manual. To support the communication to different peripherals, a generic bus multiplexer routes the memory access to the right slave peripheral. This decisssion is made up based on the memory address. To support multiple masters accessing the same peripheral, a 2:1 arbiter arbitrates the requests coming from the instruction and data interface and going to the ROM. In Figure 3.3, you find the memory map of the existing peripherals.

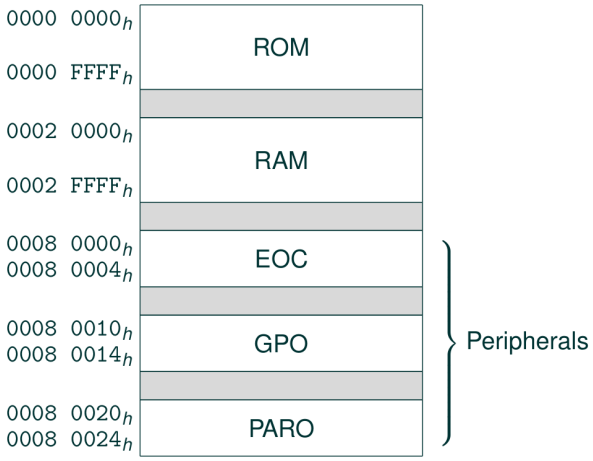

Figure 3.3: Memory map of the Ibex chip.

Software. C programs can be compiled with the toolchain installed on the cluster or locally by using the prebuild toolchain. To execute the program on Ibex, the binary is converted to a `patt` ROM file and used in the ROM module, which is instantiated in `instr_rom.sv`. The makefile located in the `ex2` directory of your repository already supports the automatic compilation and insertion into the ROM. To compile your own C program, copy and adapt the template directory `ex2/src/sw/hello-world` and adapt the `PROGRAM ?=` variable in the makefile. You can also specify the program during the call to make, *i.e.*, `make hdlsb PROGRAM=hello-world`, where the name is the folder name in the `sw` directory.

3.1 Cipher Peripheral

The cipher peripheral can be configured via a memory mapped peripheral interface. This allows the software to configure and acccess static data such as the encryption keys, nonce, or the tag. Furthermore, a control register is used to start the encryption and

determine when it finishes. To access the data, the peripheral implements a dedicated master interface to accesses the data on its own, it is implementing a DMA.

Mode of Operation. The software is used to configure and start the encryption of the cipher peripheral. First, it writes the encryption keys and nonce values to the peripheral. It configures the starting addresses for the associated data, the input data, and the output data. For example, this could be the starting addresses of C-arrays in the software. Finally, it also configures the length of the associated and the data being encrypted. Then, a control register is written indicating that the encryption operation starts. The cipher switches to a busy state and starts fetching the associated data from the configured memory address and processes it. Then, it continues with the encryption, fetching and writing the required data from the memory. Meanwhile, the software polls the status register until it is indicating that the operation has finished.

Peripheral Interface. The configuration of the cipher is memory-mapped, thus, the cipher wrapper implements a peripheral interface. Attach this interface to the existing bus multiplexer for the peripherals. Extend its size and attach the cipher peripheral on the top level interface. Map the peripheral to the address `0x80100` by using the the following configuration in `ibex.svb`.

```
`define CIPHER_START 32'h00080100
`define CIPHER_SIZE  32'h00000100
`define CIPHER_MASK  (~(`CIPHER_SIZE-1))
```

Look at the existing peripherals and the bus protocol specification regarding the implementation of the bus protocol.

DMA Data Interface. The cipher peripheral directly fetches the associated data and the plaintext and writes back the ciphertext to the data memory. To be able doing that, it must implement a DMA interface to access the data memory. In the original design, only the CPU has access to the data memory. Change the bus architecture and use a bus arbiter to arbitrate requests coming from the CPU (via the bus multiplexer) and the cipher peripheral. Within the cipher peripheral, you need to arbitrate between read requests for the associated data and the plaintext and the write requests for the ciphertext. Furthermore, the cipher peripheral needs to perform a size conversion, *i.e.*, it needs to perform multiple requests on the 32-bit bus to fetch one block of associated data, plaintext, or ciphertext.

Software Interface. Write a program that programs the cipher peripheral and performs an encryption. Configure the accelerator, program source and destination addresses and start the encryption. Poll the accellerator until it finishes. You may use your high-level model to generate test data for that program.

3.2 Specification

Your implementation must fulfill the following specification:

- Implement a cipher peripheral with a peripheral and master interface.
- Integrate the peripheral to the Ibex SoC
- Write a software that is using the cipher and showcases its functionality.
- Run the synthesis and place&route on the cluster (check for timing violations, latches, ...).
- Extend the design document from ex1 with the following points:
  - Extract the chip area (in GE) from the logs.
  - State the maximum frequency of the design.
  - Insert a picture of the chip after place&route.
  - Explain your testing strategy.
  - Explain the integration of the accelerator into the Ibex chip.

3.3 Grading

- Task 2: 40 points
  - Design: 25 points
  - Software & Test : 10 points
  - Design document: 5 points
- Deductions:
  - Coding standard: -5 points
  - Errors and warnings (e.g. latches): -5 points

3.4 Deliverables

All files must be submitted in folder `ex2` of your repository.

IMPORTANT Make sure to *commit* your files and *push* them to GitLab.

IMPORTANT Include the design document in the repository.

IMPORTANT Don't forget to create a tag and push the tags according to Section 1.3.

4 Errata

This chapter lists releases and changes of this file.

2021-03-09 Initial release.

2021-03-27 Update timing diagram.

2021-04-27 Exercise 2.

2021-05-01 Fix CIPHER_MASK definition.