FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Fast data-acquisition tools for side-channel analysis in FPGA
Student:	Bc. Ondřej Semrád
Supervisor:	Ing. Vojtěch Miškovský
Study Programme:	Informatics
Study Branch:	Design and Programming of Embedded Systems
Department:	Department of Digital Design
Validity:	Until the end of summer semester 2020/21

Instructions

Design and implement the following tools for side-channel analysis in Sakura-G board:

- configuration of control FPGA implementing communication between main FPGA and PC
- wrapper for main FPGA implementing communication with control FPGA and encryption management
- plug-in(s) for Side-Channel Analysis toolKit (SICAK) controlling the measurement process

The toolkit should:

- minimize communication with PC using data generation in control FPGA
- maximize communication speed with PC - proper FTDI mode needs to be chosen with the ability to switch between proprietary drivers and VCP
- realize PRNG suitable for random input data generation replicable in PC
- be suitable for running various ciphers using various encryption modes and using various side-channel attack countermeasures
- be suitable for running various measurement scenarios using optional randomization of each input

Implementation should be realized using VHDL (FPGA) and C/C++ (SICAK plug-in)

References

Will be provided by the supervisor.

doc. Ing. Hana Kubátová, CSc. Head of Department	doc. RNDr. Ing. Marcel Jiřina, Ph.D. Dean

Prague February 5, 2020

Czech Technical University in Prague
Faculty of Information Technology
Department of Digital Design

Master's thesis

Fast Data-acquisition Tools for Side-channel Analysis in FPGA

Bc. Ondřej Semrád

Supervisor: Ing. Vojtěch Miškovský, Ph.D.

28th May 2020

Acknowledgements

First, I would like to thank my girlfriend for her infinite love, caring and patience with my sometimes intolerable self. Then, I want to thank my supervisor for his tips, positive approach and a will to help when I was running out of time. Lastly, I would like to thank my family and my girlfriend's family for their love, caring, and their endless effort to remind me not to take this life too seriously.

This thesis was also supported by the SGS20/211/OHK3/3T/18 student grant.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In In Prague on 28th May 2020 ....

Citation of this thesis

Semrád, Ondřej. *Fast Data-acquisition Tools for Side-channel Analysis in FPGA*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

K provedení útoku odběrovou analýzou na kryptografické zařízení je třeba naměřit až miliony průběhů spotřeby tohoto zařízení. Cílem této práce je vytvořit sadu nástrojů, která urychlí a usnadní proces získávání průběhů spotřeby a zároveň bude podporovat co nejvíce různých šifrovacích algoritmů. Sada nástrojů bude zaměřená na implementace šifrovacích algoritmů v hardware, konkrétně v FPGA.

Klíčová slova Odběrová analýza, bezpečnost, FPGA, Sakura-G

Abstract

To mount a power analysis attack on a cryptographic device, one has to acquire up to millions of power traces of the attacked device. The goal of this thesis is to create a toolkit which will make the power traces acquisition faster whilst supporting as many different cryptographic schemes as possible. The toolkit will focus on hardware implentations of cryptographic schemes in FPGA.

Keywords Power analysis, security, FPGA, Sakura-G

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded devices with an integrated cryptographic algorithm are all around us. The prime example of such a device is a debit card or generally any other kind of a smart card. Since the publication of *Introduction to differential power analysis* in 1998 [5], numerous bright minds among engineers, mathematicians, and computer-security enthusiasists have been trying to come up with new sophisticated coutermeasures to secure the device against such an exploitation. Since no protection is ever perfect, the goal of the countermeasures is to make such an attack impractical — either because of the price of the tools required, or because of the unbearably long duration of the attack.

The members of the Embedded Security Lab at Faculty of Information Technology, CTU Prague, are pursuing new ways of attacking such devices, and at the same time developing new countermeasures. The goal of my thesis is to create a toolkit, which will make their effort less time consuming and thus perhaps easier.

1.1 Power Analysis Attack

Power analysis attacks belong to a more general group of attacks called the Side-Channel Attacks. These types of attack exploit the leakage of information from the cryptographic device which is not caused by a weakness in the mathematical description of the implemented cipher itself, but rather in its imperfect implementation, or in the physical properties of the device. In the case of the power analysis attack, this property is the power consumption of the device. The core idea of the Power Analysis Attact is that the immediate power consumption of the device depends on the data currently being processed.

If we have a physical access to the device, we can measure its power consumption in a way depicted in Figure 1.1. The oscilloscope measures the voltage drop across the resistor connected in series with the device. The current flowing through both the resistor and the device is proportional to the voltage as described in the Ohm's Law, and the power consumption is the product of the voltage and the current. In some cases, like in the Simple Power Analysis [6], several or even only one power trace is sufficient to perform an attack. In other cases, like in the Differential Power Analysis [6], the input or the output data of the encryption device are also required.

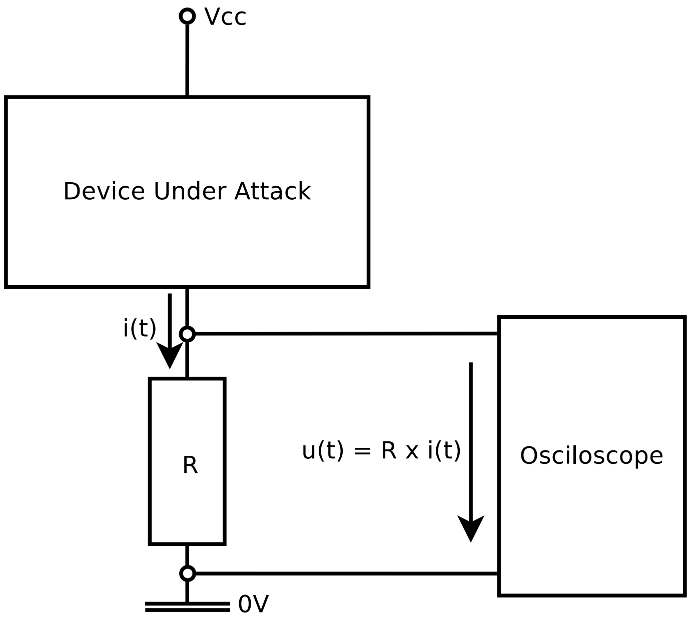

Figure 1.1: An example of a power measurement setup.

- **Differential Power Analysis with Difference of Means distinguisher** — We choose an intermediate value – a bit – in the device that depends both on the input key (unknown) and the data (known) in a non-linear way. We choose a small part of the key (e.g. a byte) and precompute the intermediate value for every possible key value and input/output data combination. Then, we separate the measured traces into two groups depending on the intermediate bit value and compare the groups. We calculate the mean vectors of the two groups and substract them — when the correct key hypothesis is made, a significant peak value shall occur in the calculated difference vector, given that we have sufficient number of power traces.

- **Differential Power Analysis with Correlation Coefficient distinguisherem, also known as Correlation Power Analysis (CPA)** — We choose a power consumption model — Hamming weight is commonly used; Hamming distance is more precise but it requires two intermediate values. We choose an intermediate value(s) – a byte(s) – in the device that depends both on the input key (unknown) and the data (known) in a non-linear way. We choose a small part of the key (e.g. a byte) and precompute the intermediate value(s) for every possible key value and input/output data combination. We compute power predictions by applying the chosen model on the intermediate value(s). The power consumption predictions are then correlated with the measured power consumption traces, and if the key hypothesis was correct, there shall be a significant peak value in the correlation vector, given that we have sufficient number of power traces. CPA was first introduced in [1].

1.1.1 Statistical Evaluation

To get a degree of certainty that the DPA attack was successful, and not to have to rely on the plain "visibility" of the peak value in the difference vector, Welch's unequal variances T-Test [14] can be used to test the null hypothesis that the means of the two groups of the power traces are equal. The T-Test is conducted at each sample point separately, resulting in a vector of p-values. This type of the statistical evaluation of the information leakage of the device is called *Specific T-Test* and it is described in more detail in *Leakage Assessment Methodology* [10]. It is called *Specific T-Test*, because it relies on a specific intermediate value in the design of the cryptographic device. There might be numerous such values, and performing the attack using every one of them would be very impractical and time consuming. Hence, the *Leakage Assessment Methodology* proposes another method of testing the information leakage — *Non-Specific T-Test*, also called *Fixed vs. random test.* In this test scenario, two groups of power traces are measured — one using constant data and one using random data. These two groups are then again evaluated using the unpaired T-Test, and shall any leakage occur in the design, the resulting p-value vector shall contain a peak value. The fixed and random data shall be also measured in a random order to ensure that the device is in a random state prior to random measurement, otherwise the *Non-Specific T-Test* could report a non-existing leakage.

1.1.2 Countermeasures

Countermeasures against the described types of attacks are generally split into two main groups — hiding and masking. Hiding tries to hide the information either in time, e.g., by inserting random delays, or hide the power consumption by using a device design whose power consumption is independent of the data being processed. Masking applies one or multiple random bit-vectors to the sensitive data to randomize the power consumption. The cipher itself has to be modified to handle the modified data and to calculate the correct encrypted values. *Non-Specific T-Test* can be also applied to the power traces of a device which uses the masking techniques, but the power traces have to be preprocessed first [10].

The common feature of the described attacks is that many power traces have to be measured, especially for a device with countermeasures. Such a measurement can be a lengthy process, and the goal of my thesis is to create a set of tools which will make the data acquisition during such an attack faster and also more convenient.

Chapter 2

System Analysis

In this chapter, the FPGA board to be used is introduced, the analysis of the system requirements is done and based on the analysis a specification of the final system is created.

2.1 Sakura-G

Sakura-G board is a device designed in Japan's Morita Tech Company specifically for the Side-Channel Attacks [8]. The key attributes, which make the board excellent for power analysis attacks, are:

- Two Xilinx Spartan-6 FPGAs with separate voltage regulators. The larger FPGA – Main FPGA, Xilinx XC6SLX75-2CSG484C – is meant for the cipher implementation and its power consumption measurement. The smaller FPGA – Control FPGA, Xilinx XC6SLX9-2CSG225C – is meant as a data preprocessor for the Main FPGA. The FPGAs are interconnected by a 51 bits wide bus.

- Pre-installed 1Ω resistor connected in series with the Main FPGA power line in a similar manner as in Figure 1.1. Custom resistor can be inserted in parallel with the pre-installed one to get the desired resistance using the jumper JP2.

- Pre-installed measurement points — SMA connectors J1, J2 and J3. The signal measured on connector J3 is already pre-amplified by 20dB using the built-in AD8000 amplifier.

- Two 48 MHz oscillators, one for each FPGA. The clock source can be easily replaced with a custom one using the J4, J5, J6 and J7 SMA connectors.

- USB communication interface chip FT2232H [3].

The top view of the board is in Figure 2.1.

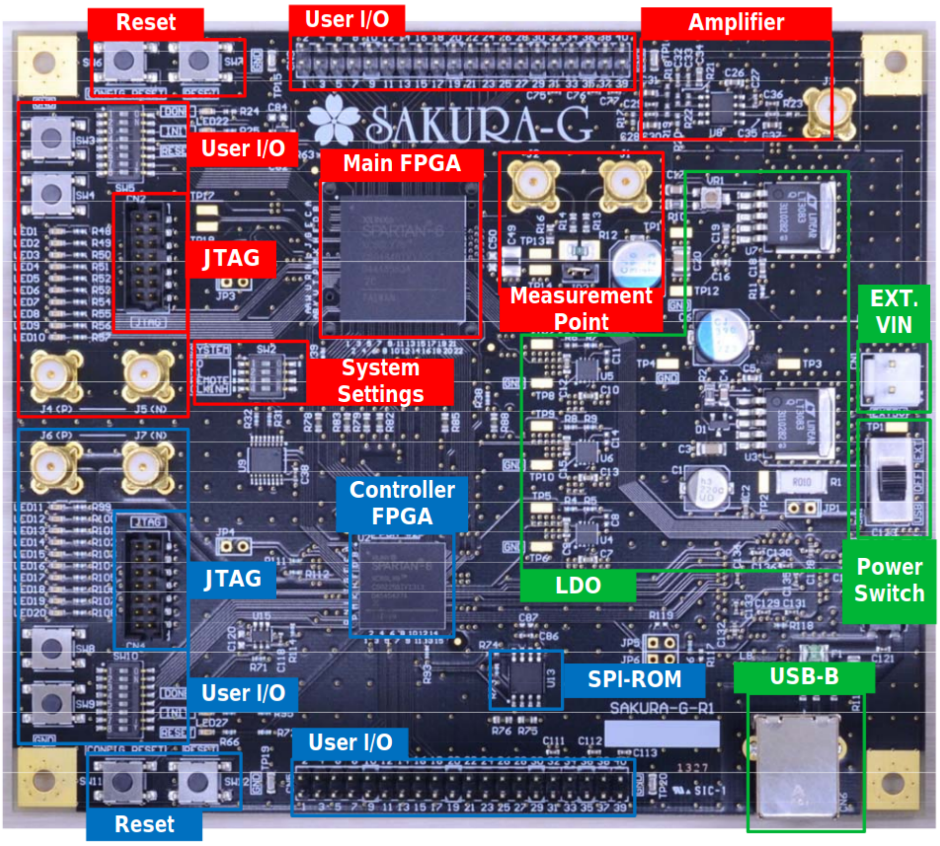

Figure 2.1: Top view of the Sakura-G board [8].

2.2 System Overview

As specified in the thesis assignment, the final system should use the Control FPGA of the Sakura board to minimize the communication with the computer. This is the core idea of my thesis and it was originally presented in *Leakage Assessment Methodology* [10] — setup described in the article uses "Control" to generate input values for the "Target". Since these entities fit exactly into the purpose of the Control and the Main FPGA of the Sakura-G, I call them Control and Main throughout the thesis. The global overview of the system is in Figure 2.2.

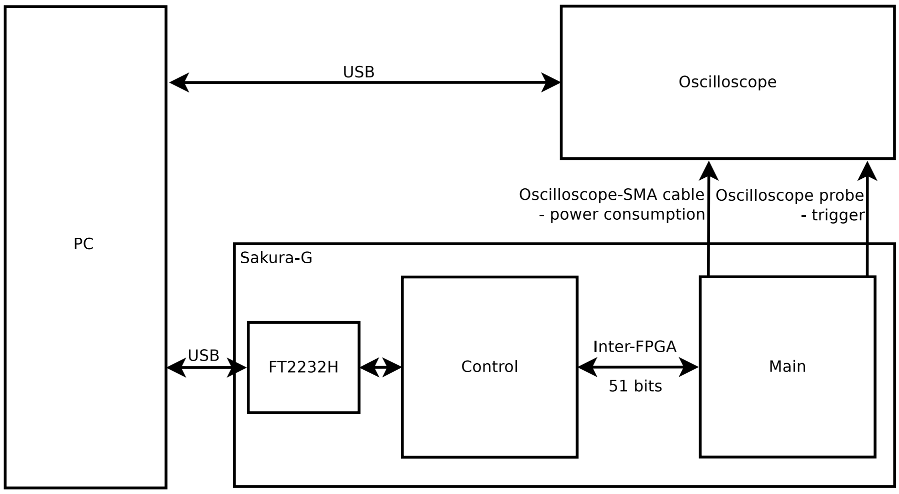

Figure 2.2: Block diagram of the complete system. The probe of the oscilloscope measures voltage drop across the resistor connected in series with the Main FPGA.

The following requirements are in the thesis assignment:

- **Minimize the communication with the computer using data generation.** — As described in [10], the computer shall only send the number of measurements and a seed, and the Control will generate all the data required by the cipher automatically. A good practice is to implement a way to check that no error occured during the data generation — a checksum of the computations, that is sent back to the computer at the very end of the operation. The input/output data sending (usually realized using UART) is the main performance bottleneck of the current power measurement setup used by the members of the Embedded Security Lab of FIT CTU.

- **Maximize the communication speed with the computer by using the proper mode of FT2232H.** — FT2232H [3] supports UART mode, where the data received from the computer over USB are converted into RXD signals and the data received from the device over TXD are sent back to the computer using the USB protocol. This mode uses Virtual Com Port (VCP) drivers and the software in the PC can therefore use the device as a standard COM port. The second option is the synchronous FIFO mode — the proprietary D2XX drivers of FTDI are required and a custom application has to be created to use this driver properly.

- **Support various ciphers using various encryption modes and using various side channel attacks countermeasures.** — Since the cipher is seen as a black box by the device, the only part of interest of the cipher is its interface. Various ciphers have different input and output data widths; ciphers using the masking countermeasure techniques might accept the data in the shared form. Hence, the device shall be able to generate input data using various widths and various numbers of shares, and it shall be able to handle the output data with the same variable properties.

- **Realize PRNG suitable for input data generation replicable in PC.** — The PRNG should support various data widths as described in the last item, and it shall be deterministic, so its behaviour can be replicated in the computer.

- **Be suitable for running various measurements scenarios using optional randomization of each input.** — The attacks mentioned in Chapter 1 require three different measurement scenarios:

  - Random — The cipher's input changes after every encryption (DPA).
  - Constant — The cipher's input does not change (SPA; Signal-to-noise ratio measuring).
  - Fixed vs. random — The cipher's input is either constant or random in a random order (*Non-Specific T-Test*)

Given the requirements, the following functionalities will be needed in the elements of the final system.

2.2.1 Control

Control should function as a middle point between the computer and Main. It should implement a memory to store the initialization data and the immediate input and output values. Multiple operation modes shall be supported for every input. In order for the device to support as many different encryption scenarios as possible, an operation code should be received among the initialization data for every input separately. The Control should implement a PRNG scheme to optionally randomize the data. To ensure that the data at the cipher's input are random before every encrytion, and to ensure that the cipher gets fresh masks before every encryption, the Control should implement a scheme to remask the data. It shall implement module to communicate with the computer. Different modules will be required for the UART and for the Synchronous FIFO mode of FT2232H. Lastly, it must implement a module for data sending to/data receiving from Main. All of the functionalities shall be supported for any width and any number of shares of the input.

2.2.2 Main

Main should function as a wrapper of a cipher which receives the data from Control. It should implement a module for communication with Control and a memory to save the received input data. It shall implement an encryption management — it should start the cipher when the inputs are ready and wait for its runtime to end. It shall also implement the trigger signal generation for the oscilloscope.

2.2.3 Computer

The computer application should send the initialization data to Control and be able to replicate the exact cipher's inputs depending on the initialization data sent. The output data should be received from Control, or they can be calcualted in the application. It should also manage the oscilloscope. At the end of the operation, the application should save the measurement data.

2.2.4 Pseudo-random number generator

To generate a random bit-vector of a variable length, multiple instances of the same PRNGs running in parallel can be used. Although not ideal, it shall suffice the requirement to generate random cipher inputs or to generate new masks for the inputs in the shared form. Millions of power traces might be required for attacking a masked cipher [10], therefore a 32 bit wide PRNG with $2^{32} - 1$ unique values should be more than sufficient. Multiple PRNG options exist, but the linear-feedback shift register is the best option for a hardware design for its low area requirements.

2.3 FPGA Design Specification

Given the design of the Sakura-G board, the Control FPGA design shall implement most of the device's functionalities and the Main FPGA design shall only serve as a small sub-wrapper of the cipher itself. Design requirements resulting from the analysis that we agreed on with the thesis supervisor follow:

- Arbitrary number of cipher's inputs and outputs shall be supported.
- Arbitrary widths of cipher's inputs and outputs shall be supported.
- Following modes of operation shall be supported for every input independently:
  - Fixed mode – the input never changes.
  - Random mode – the input is randomized before every cipher run.
  - Fixed vs. random – either random input or a fixed one is put at cipher's input in a random order.

  - Received from PC – the input is sent from the computer before every cipher run.

- Inputs' initial values shall be generated using pseudo-random number generator or received from the computer.

- Inputs in shared form shall be supported. The number of shares is arbitrary. When the shared form is used, unmasked data shall never occur in the design — the data shall be received from PC in the shared form and passed to the cipher in the shared form.

- The design shall be able to generate new masks pseudo-randomly and re-mask the input using the new masks.

- The device shall be able to send cipher's outputs back to the computer.

- The device shall optionally generate cipher's random input. This input shall provide fresh randomness every clock cycle.

- The device shall generate a trigger signal for the oscilloscope. There shall be a delay of an adjustable length between trigger-on and cipher-start. There shall also be a delay of an adjustable length between cipher-done and the continuation of operation. The trigger shall be on either for one clock cycle, or for the whole duration of delays and the cipher runtime.

Turning described options on/off or changing their parameters can be theoretically done from the computer at runtime, but given the use-case of the final device, we agreed on the following requirements:

- Widths of the inputs and the outputs and their share count shall be known prior to the synthesis.

- Mode of operation shall be settable at runtime.

- Remasking feature shall be settable at runtime.

- Cipher's random input presence and seed shall be set before the synthesis.

- The trigger mode and the optional delays before and after the cipher runtime shall be set prior to the synthesis.

Functionalities adjustable at runtime shall be set using operation codes. The device will receive operation codes in the initial phase of every run. Every input and every output of the cipher shall have its own operation code. Input operation code's structure is in Table 2.1.

Bit nr.	Description
0 - 3	Mode of operation.
4	Remask off/on bit (0/1).
5	Initialize input randomly/from computer bit (0/1).
6 - 7	Unused.

Table 2.1: Input operation code structure.

The mode coding is in the Table 2.2.

Bit value	Description
0000	Fixed mode.
0001	Random mode.
0010	Fixed vs. random mode.
0011	Sent from PC mode.
Others	unused

Table 2.2: Input operation code mode code table.

Output operation code's structure is in Table 2.3.

Bit nr.	Description
0	Do nothing/sent to computer bit (0/1).
1 - 7	Unused.

Table 2.3: Output operation code structure.

The last requirement is that the run of the device shall be deterministic and thus everything shall be replicable in the computer with the only exception being the cipher's random input.

2.4 Software Specification

As mentioned in the assignment, Side Channel Analysis Toolkit (SICAK) [11] should be used to implement the software part of the system. SICAK is written in C++ programming language and it uses Qt5 framework [12]. SICAK is modular, and the utility used for side-channel measurement is called *meas*. A measurement-scenario plugin must be created to support the proposed measurement setup. The plugin shall support the same set of functionalities as specified in Section 2.3, but from the opposite side — where the FPGA design receives data from the computer, the plugin has to send them and vice-versa. Every input of the cipher must be calculated in the plugin to be later saved into a file. The plugin shall be prepared for the insertion of a software cipher implementation to calculate the outputs and not to depend solely on their receiving from the device.

SICAK already supports communication over a serial port using the SerialPort plugin. It also supports two types of oscilloscopes. For the D2XX proprietary drivers of FTDI, an additional communication plugin has to be created. We have agreed with the thesis supervisor that the support for this mode of FT2232H operation will not be implemented, since the purpose of the device to be implemented is to remove the necessity for most of the communication, and the communication speed will then not present a significant problem.

Chapter 3

FPGA Design and Verification

In this chapter, the design specification and verification are described.

3.1 Design Implementation

The design consists of two entities - Control and Main - interconnected by four handshake signals and two simplex buses. Global overview of the design is in Figure 3.1.

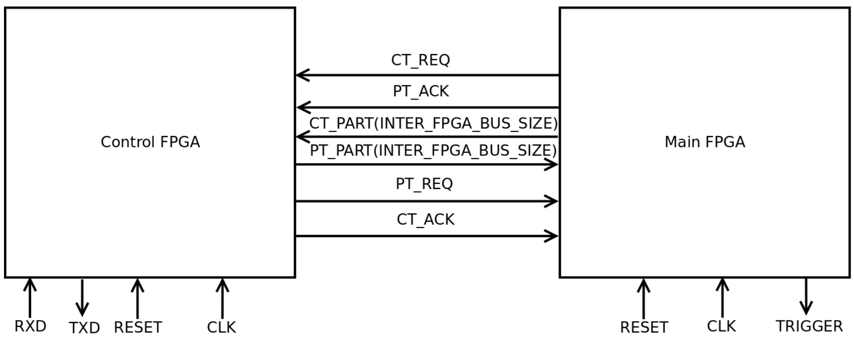

Figure 3.1: Global overview of the design.

Sakura-G uses different clock sources for its FPGAs, therefore a handshake communication protocol was chosen for data passing between Main and Control. The protocol is described in Figure 3.2.

PT_PART and CT_PART buses data validity during sampling is ensured by the handshake protocol. The handshake signals are synchronized at entities' inputs using a synchronizer, so that no metastability problems can occur [2].

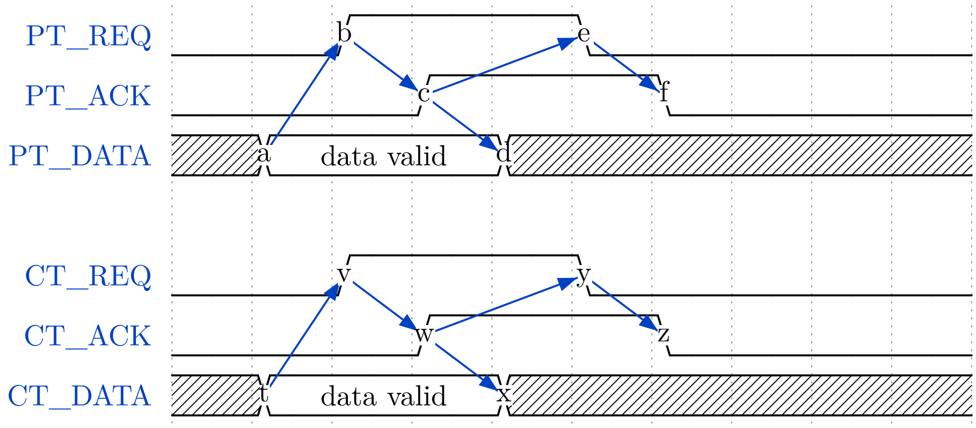

Figure 3.2: Handshake protocol. Arrows describe the necessary order of the changes of the handshake signals.

3.1.1 Common Design Elements

In this section, the design elements used by both Control and Main are presented.

DEFINITIONS package contains declarations of all the constants required by the design. It also contains declarations and definitions of some types, functions and procedures. The list of important constants of the package is in the table 3.1.

Although possible, other constants in the package are not meant to be modified by hand. There are hidden dependencies between various constants and hand modifications are prone to error. These ought to be generated automatically using the *generate_definitions.py* script. The source of information for this script is the *config.txt* file – see the file for details, it contains explanatory commentaries. Input and output count, individual input and output widths, number of shares, etc., shall be filled into *config.txt* file. Testbench related constants are also generated automatically using the *verify.py* script. More info about *verify.py* follows in the Section 3.2.

REG entity is used everywhere, where a block of data has to be stored. The entity interface is in Table 3.2.

Name	Description
CLK_F	Clock frequency assigned to Control's *CLK_ORIG* input port.
BAUD_RATE	UART baud rate.
INTER_FPGA_BUS_SIZE	Width of one of the two inter-FPGA buses.
LFSR_WIDTH	Width of the Linear-Feedback Shift Register used as PRNG's basic element.
N_COUNTER_WIDTH	Width of the counter counting number of cipher repetitions remaining.
OP_REG_SIZE	Width of the input and output operation code.
TIMER_THRESHOLD	Minimum number of clock cycles between individual cipher's runs.
*_MPX_SEL_WIDTH	Widths of various multiplexers' select signals in the design.

Table 3.1: List of constants in DEFINITIONS package to be modified by hand.

Generics		
SIZE	width of the register	
INPUT_SIZE	width of the input port	
OUTPUT_SIZE	width of the output port	
Ports		
CLK	in	clock input
RESET	in	reset input
INPUT	in	data input, connected to INPUT_SIZE bottom bits
LOAD	in	shift left by INPUT_SIZE and load INPUT bits to bottom
SHIFT	in	shift left by OUTPUT_SIZE and fill bottom bits with zeroes
OUTPUT	out	data output, connected to OUTPUT_SIZE top bits

Table 3.2: REG entity interface.

An example is also depicted in Figure 3.3. *REG* is a shift register with two shift lengths, input connected to the bottom bits and output connected to the upper bits. An extreme case would be an instance of this entity with all three generic parameters equal; then the full register length would be connected to the whole *INPUT* and *OUTPUT* signals, *LOAD* signal would fill the register with data, whereas *SHIFT* signal would fill the register with zeroes.

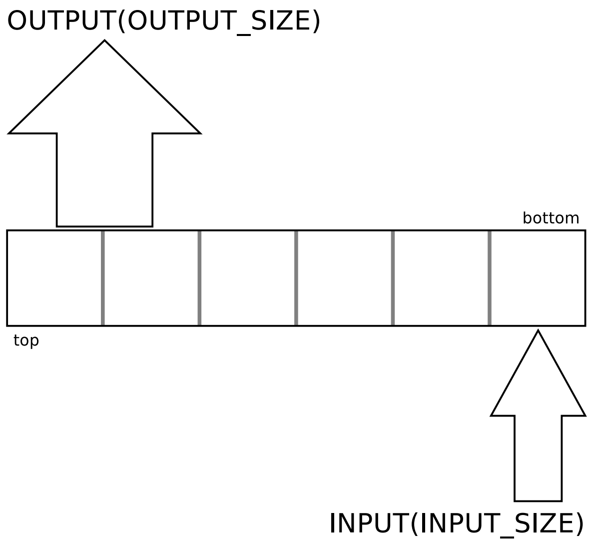

Figure 3.3: Block diagram of an example instance of the REG entity. In this example, the OUTPUT_SIZE is two times the INPUT_SIZE and WIDTH is six times the INPUT_SIZE.

MULTIPLEXER is a pure combinational circuit that implements 8 to 1 multiplexing. Interface is in Table 3.3.

Generics		
WIDTH	width of inputs and output	
Ports		
INPUT{0..7}	in	inputs
SEL	in	select signal
OUTPUT	out	output

Table 3.3: MULTIPLEXER entity interface.

This entity is used where data multiplexing occurs. When the number of required input ports is smaller than eight, remaining input ports and unused select signal bits are connected to logical zero and will get optimized away during the synthesization process.

SYNCHRONIZER synchronizes signals comming from a different clock domain, so that no metastability can occur [2]. It is implemented as two D flip-flops in series. Its interface is in Table 3.4.

Ports		
CLK	in	clock input
RESET	in	reset input
INPUT	in	input
OUTPUT	out	output

Table 3.4: SYNCHRONIZER entity interface.

This entity is used for synchronization of the request/acknowledge signals on the inter-FPGA bus.

COUNTER counts from the initially set value down to zero. Its interface is in Table 3.5.

Generics		
WIDTH	width of the internal counter signal	
Ports		
CLK	in	clock input
RESET	in	reset input
SET_VALUE	in	value to be set to the internal counter signal
SET	in	when '1', set SET_VALUE to the internal counter signal
ENABLE	in	when '1', the internal counter value is decremented by one
VAL	out	value of the internal counter signal
CNT_DONE	out	'1' when the internal counter signal value is zero

Table 3.5: COUNTER entity interface.

All counting and timing instances in the design are instances of *COUNTER* — most notably *N_CNT*, counting the number of encryption repetitions; *REG_CNT*, counting over all inputs or outputs and *SHARES_CNT*, counting over individual input's or output's shares.

INTER_FPGA_COMM implements low-level data handling during sending to/receiving from the other FPGA. Its interface is in Table 3.6.

Generics		
INPUT_DATA_SIZE	maximum width of the data to be received	
OUTPUT_DATA_SIZE	maximum width of the data to be sent	
INTER_FPGA_BUS_SIZE	width of the one-way data buses between Main and Control	
CNT_WIDTH	width of the internal data counter	
CNT_SKIP_WIDTH	width of the internal skip counter	
Ports		
CLK	in	clock input
RESET	in	reset input
CNT_SET_VALUE	in	value to be assigned to the internal data counter (counting the data to be sent/received)
CNT_SKIP_SET_VALUE	in	value to be assigned to the internal data counter (counting the data to be skipped before send)
SEND	in	signal telling the module to start sending
RECEIVE	in	signal telling the module to start receiving
SENT	out	signal indicating that sending is over
RECEIVED	out	signal indicating that receiving is over
INCOMING_DATA	out	received data
OUTGOING_DATA	in	data to be send
SEND_REQ	out	send request
SEND_ACK	in	send acknowledge
SEND_DATA	out	one-way send data bus
REC_REQ	in	receive request
REC_ACK	out	receive acknowledge
REC_DATA	in	one-way receive data bus

Table 3.6: INTER_FPGA_COMM entity interface.

Datapath of the entity consists of two *REG* instances and is shown in Figure 3.4.

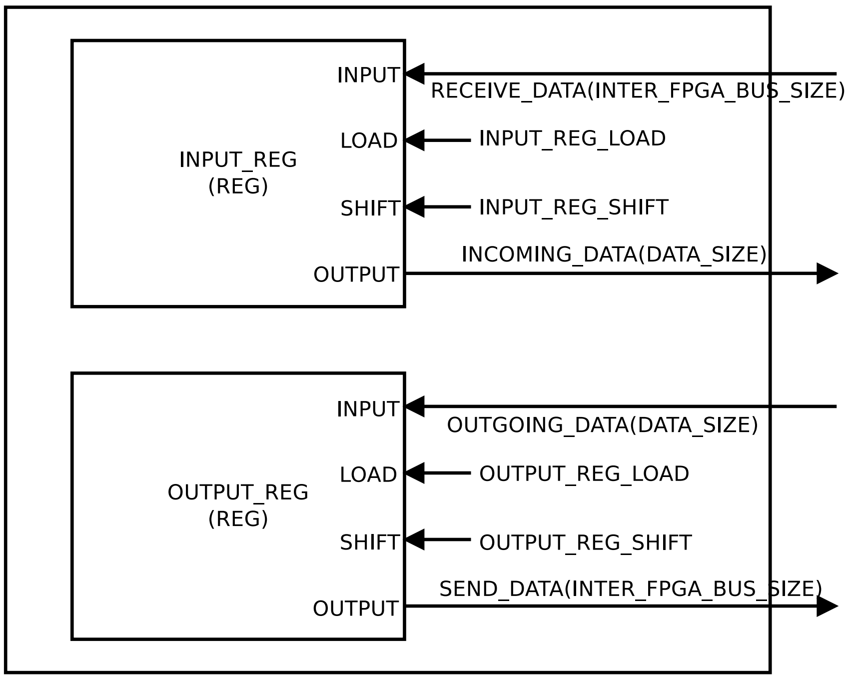

Figure 3.4: Block diagram of the INTER_FPGA_COMM datapath.

The controller of the entity is in Figure 3.5. The receive and send cycles of the automaton are actually independent, but simultaneous receiving and sending shall never occur in the design, therefore they are realized in one FSM. During sending, data skipping occurs beforehand — not all of the data to be sent necessarilly have the full *OUTPUT_DATA_SIZE* width, therefore given the design of *REG* entity, extra bits at the top have to be shifted-out first. The rest of the automaton implements the handshake protocol as described in Figure 3.2.

LFSR32 is a combinational circuit implementing the [32, 22, 2, 1] linear-feedback shift register as described in [4]. Its interface is in Table 3.7.

Ports		
INPUT	in	input
OUTPUT	out	output (input shifted one time)

Table 3.7: LFSR32 entity interface.

LFSR32 is used in *PRNG* entity, which serves as its wrapper.

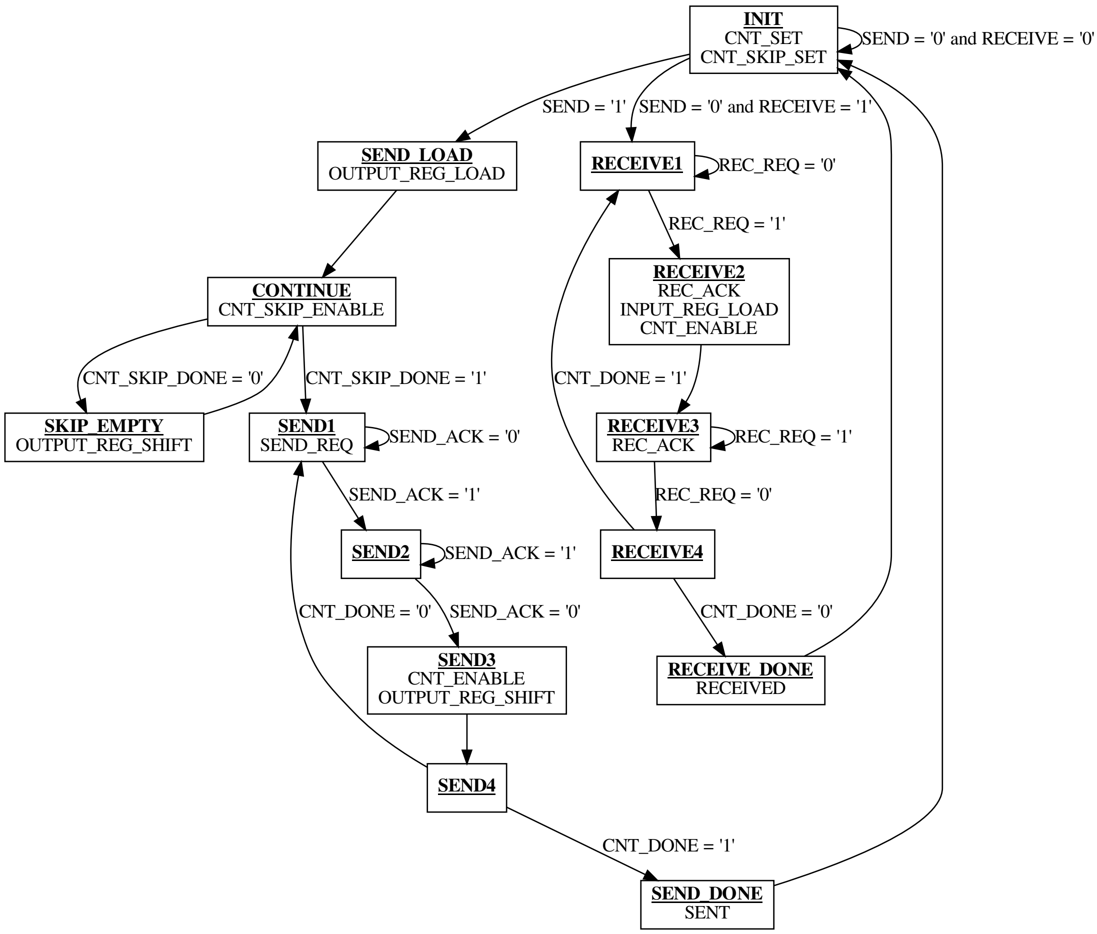

Figure 3.5: INTER_FPGA_COMM controller FSM diagram.

PRNG entity uses *LFSR32* instances running independently in parallel to create pseudo-random values of the given length. Its interface is in Table 3.8.

Generics		
SIZE	width of the generated number; shall be $32 \times k$, where k is natural	
Ports		
CLK	in	clock input
RESET	in	reset input
PRNG_SET	in	when '1', set INPUT to the internal data signal
PRNG_ENABLE	in	when '1', the internal data signal is shifted one time
INPUT	in	value to be set to the internal data signal = seed
OUTPUT	out	value of the internal data signal shifted one time

Table 3.8: PRNG entity interface.

There are three instances of the generator. The first is the *DATA_PRNG* in Control — it uses *LFSR32* instances running in parallel to match the *INPUT_WIDTH*. Some bits of *DATA_PRNG* might be extra — these are ignored in assignments. Another instance is the *CONTROL_REG* in the Control's controller. This one uses just one instance of *LFSR32* and its bit nr. zero is used for the decision of which input shall be sent to Main in Fixed vs. random mode. The last instance is in Main — this one uses shift registers in parallel again to create the random input of the cipher for its potential inner remasking needs. It is seeded in the first state of the Main's controller FSM by a constant and then it shifts every clock cycle (its *PRNG_ENABLE* input is always logical one).

3.1.2 Control

The Control is used to communicate with PC, to prepare the cipher input data and to handle the cipher output data. The summary of functionalities implemented in Control is:

- **Receive initialization data** — input operands, output operands, number of encryptions, data seed, and controller seed.

- **Receive input data from PC** — data can be received at the beginning of the operation to initialize both fixed and random inputs or before every encryption to initialize the random input.

- **Generate input data** — data are generated using the data PRNG; both random and fixed data can be generated at the beginning of the operation.
- **Randomize random input data** — data are randomized using the data PRNG.
- **Remask input data** — new masks are generated by data PRNG; random or fixed data are remasked based on the mode of operation.
- **Send input data to Main** — send random or fixed data depending on the operation mode.
- **Receive output data from Main.**
- **Send output data to PC.**

These functions combined in the right order realize the following modes of operation for every cipher's input:

- **Fixed mode** — the input remains constant during the whole operation.
- **Random mode** — the input is randomized before every encryption.
- **Fixed vs. random mode** — the input is random or fixed in the pseudo-random order. The decision is made using the control PRNG.
- **FTDI mode** — the input is sent from the computer before each and every encryption.

Control's interface is in Table 3.9.

Ports		
CLK_ORIG	in	clock input; it is assigned directly to CLK signal, which is used as the clock source for sub-entities; optional clock divisor can be inserted between CLK_ORIG and CLK
RESET	in	reset input
RXD	in	UART RXD signal input
PT_ACK	in	data send to Main acknowledge
CT_REQ	in	data receive from Main request
CT_PART	in	data bus from Main
TXD	out	UART TXD signal output
PT_REQ	out	data send to Main request
CT_ACK	out	data receive from Main acknowledge
PT_PART	out	data bus to Main

Table 3.9: TOP_CTRL entity interface.

The design itself is split into two parts – the datapath and the controller.

3.1.2.1 Control's Datapath

Global overview of the Control's datapath is in Figure 3.6.

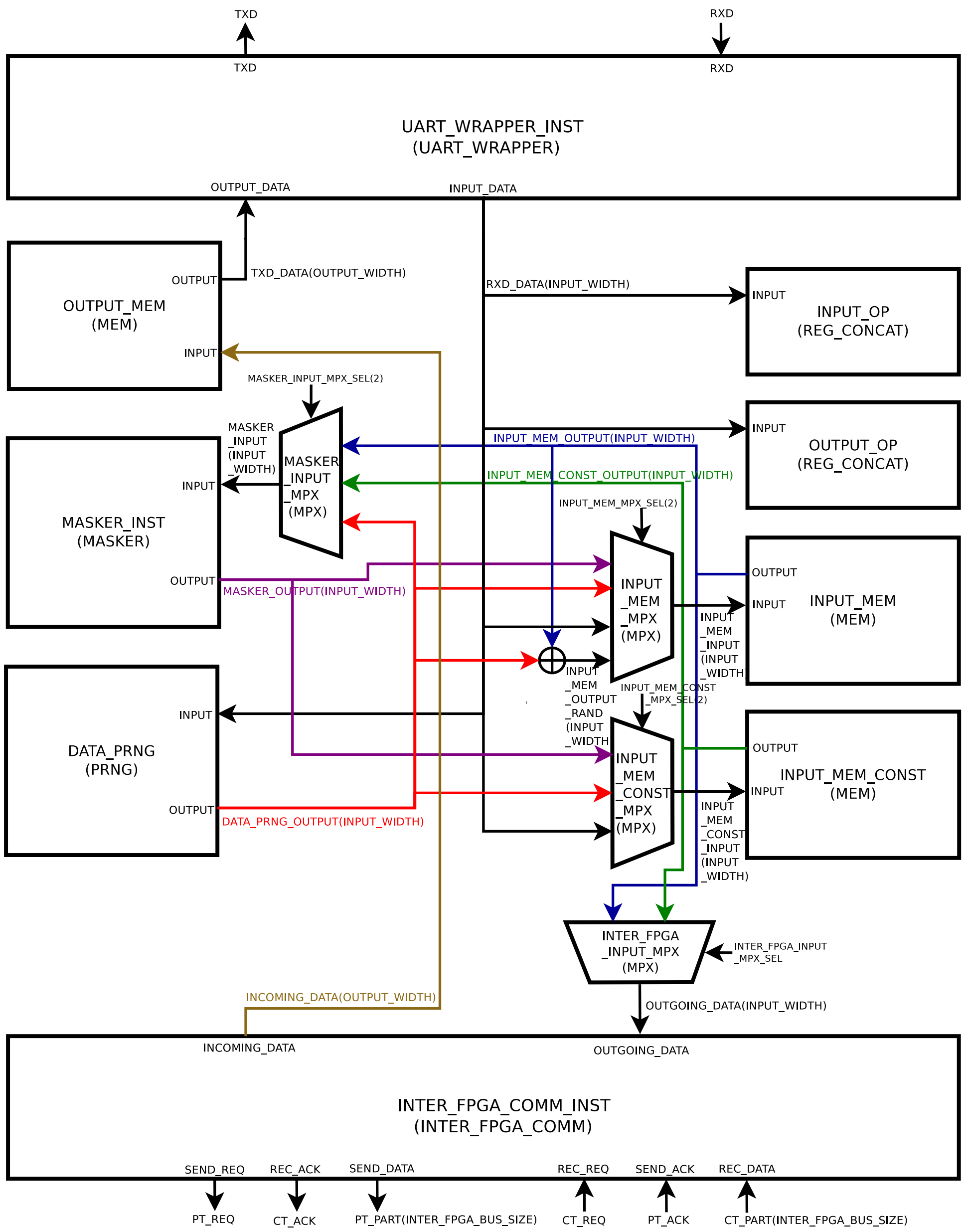

Figure 3.6: Block diagram of Control's datapath. Clock, reset and most of the control signals are skipped for better readibility.

UART_WRAPPER_INST handles the lower layers of the communication with the computer. *INPUT_OP* and *OUTPUT_OP* store 8-bit operation codes for every input/output separately. *INPUT_MEM* stores the cipher input data which are to be changed variously during operation. *INPUT_MEM_CONST* stores the fixed input data which are created before entering the main encryption loop of controller's Main FSM and never change upon entering the loop. *OUTPUT_MEM* stores the output data received from Main. These data can be either sent to the computer or ignored — their presence might be useful shall the design be ever modified to contain, e.g., a fault-free run check. *DATA_PRNG* generates pseudo-random bit vectors for random memory fills or remasking purposes. *MASKER_INST* contains the space to load the complete input from one of the input memories and the logic required to remask it without the unmasked value actually appearing anywhere. *INTER_FPGA_COMM_INST* implements the low-level communication with Main. *INTER_FPGA_INPUT_MPX_SEL* signal decides, whether the *INTER_FPGA_COMM_INST* shall send the random data from *INPUT_MEM* (logical zero) or the fixed data from *INPUT_MEM_CONST* (logical one). Its value is assigned to:

- Logical one, when in the Fixed mode.
- Logical zero, when in the Random mode or in the FTDI mode.
- Bit nr. zero of *CONTROL_PRNG*, when in the Fixed vs. random mode.

CONTROL_PRNG is enabled once before every encryption, thus inputs with Fixed vs. random mode will all have the random or the fixed input at the same time. The instantiated components are noted under instances' names and more information about them is in the following paragraphs.

UART_WRAPPER implements low-level data handling during sending to/receiving from the computer using the UART mode of the FT2232H. Its interface is in Table 3.10.

Generics		
INPUT_DATA_REGS_SIZE	maximum width of the data to be received	
OUTPUT_DATA_REGS_SIZE	maximum width of the data to be sent	
EXP_CNT_WIDTH	width of the internal data counter and skip counter	
UART_SIZE	width of data transmitted in one UART transaction, i.e. 8	
Ports		
CLK	in	clock input
RESET	in	reset input
TXD	out	UART TXD line
RXD	in	UART RXD line
READY	out	signal indicating that the TXD line is ready for sending
SEND	in	when '1', start sending of data on OUTPUT_DATA port
RECEIVED	out	signal indicating that receiving is over
EXPECTED_CNT	in	number of bytes to be sent/received
SKIP_CNT	in	number of bytes to be skipped before sending
INPUT_DATA	out	received data
OUTPUT_DATA	in	data to be send

Table 3.10: UART_WRAPPER entity interface.

The datapath of the entity consists of two *REG* instances and the UART entity instance. The UART entity was created by Dr.-Ing. Martin Novotny at FIT CTU and I modified it to use the standard IEEE *std_numeric* package instead of the proprietary *std_logic_arith* package. The datapath is shown in Figure 3.7.

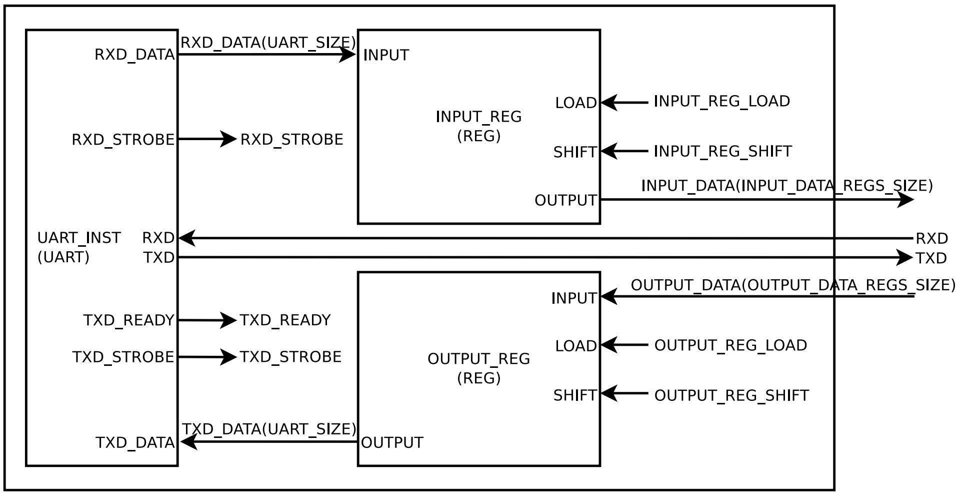

Figure 3.7: Block diagram of the UART_WRAPPER datapath.

The controller of the entity consists of two parts - RXD and TXD FSMs. The diagram of the RXD FSM is in Figure 3.8. It waits for the *RXD_STROBE* of the *UART* entity and when the signal goes high, *EXPECTED_CNT* bytes are received. End is indicated on the *RECEIVED* output port.

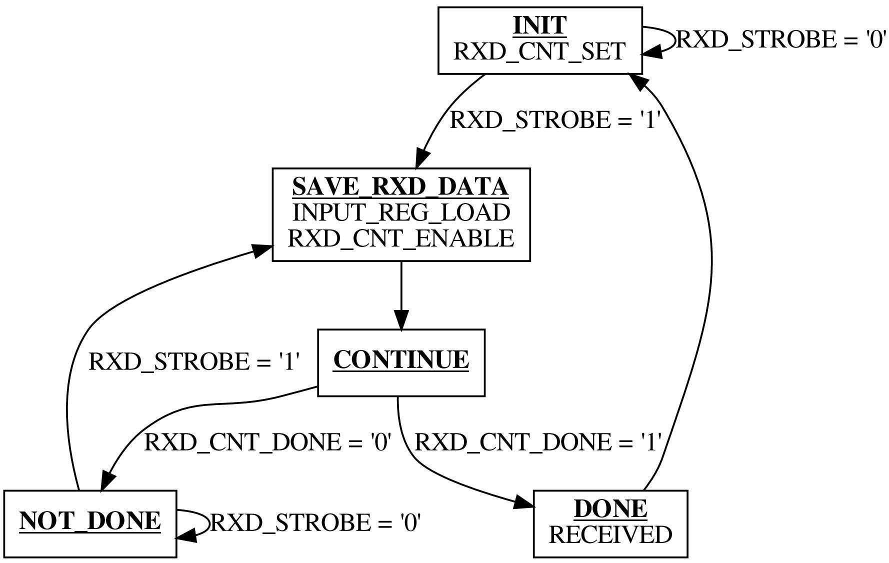

Figure 3.8: UART_WRAPPER controller RXD FSM diagram.

The diagram of the TXD FSM is in Figure 3.9. It waits for the *SEND* signal and when it goes high, *SKIP_CNT* bytes are skipped, because of the

REG entity design. After that, *EXPECTED_CNT* bytes are sent to the TXD output port.

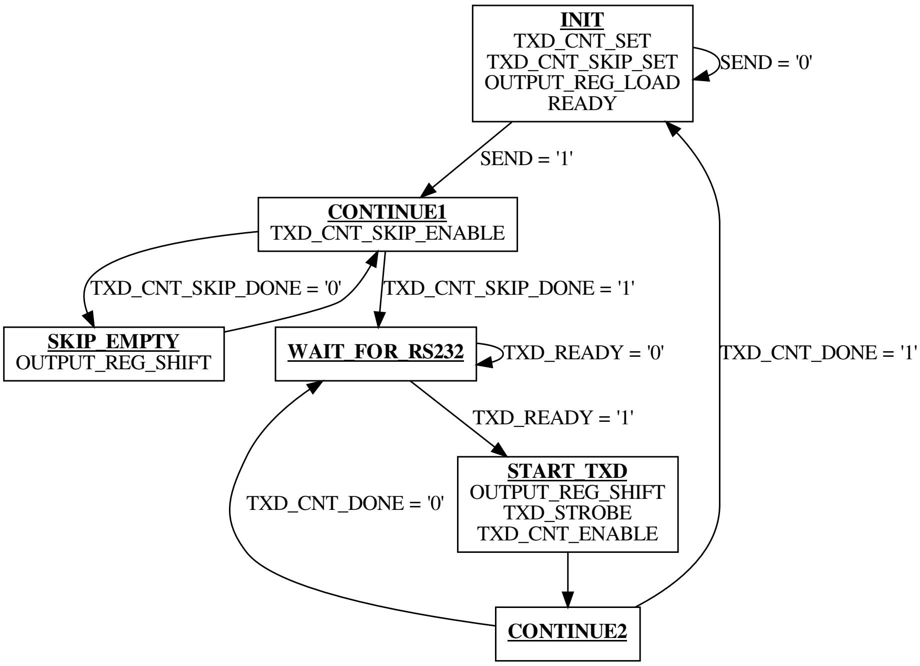

Figure 3.9: UART_WRAPPER controller TXD FSM diagram.

REG_CONCAT implements addresable 1D array of *REG* instances. *LOAD* port of *REG* component is used to load new data; *SHIFT* is assigned to logical zero. Its interface is in Table 3.11.

The entity is also depicted in Figure 3.10.

INPUT port is connected to every register instantiated inside. To control the loading of registers, a load signal of array type (*LOAD_AR*) is used. *LOAD_DMPX* block in Figure 3.10 is implemented as a process in VHDL to support an arbitrary number of instantiated registers. *OUTPUT_MPX* is implemented as an array-type element access for the same reason.

REG_CONCAT is instantiated as *INPUT_OP* and *OUTPUT_OP* entities. It is also used in *MEM* entity as a 1D sub-array instance.

MEM implements an addressable 2D memory array created as a 1D array of *REG_CONCAT* instances. Its interface is in Table 3.12.

The entity is also depicted in Figure 3.11.

INPUT port is connected to every *REG_CONCAT* instance. To control the loading of registers, a load signal of array type (*LOAD_AR*) is used. *LOAD_DMPX* block in the Figure 3.11 is actually implemented as a process in VHDL to support an arbitrary number of instantiated *REG_CONCAT*

Generics	
REG_CNT	number of registers in the array
REG_WIDTH	width of every register in the array
ADDR_WIDTH	width of the address signal

Ports		
CLK	in	clock input
RESET	in	reset input
INPUT	in	data to be saved to the addressed register
LOAD	in	when '1', save INPUT to the addressed register
ADDR	in	address signal
OUTPUT	out	value of the currently addressed register

Table 3.11: REG_CONCAT entity interface.

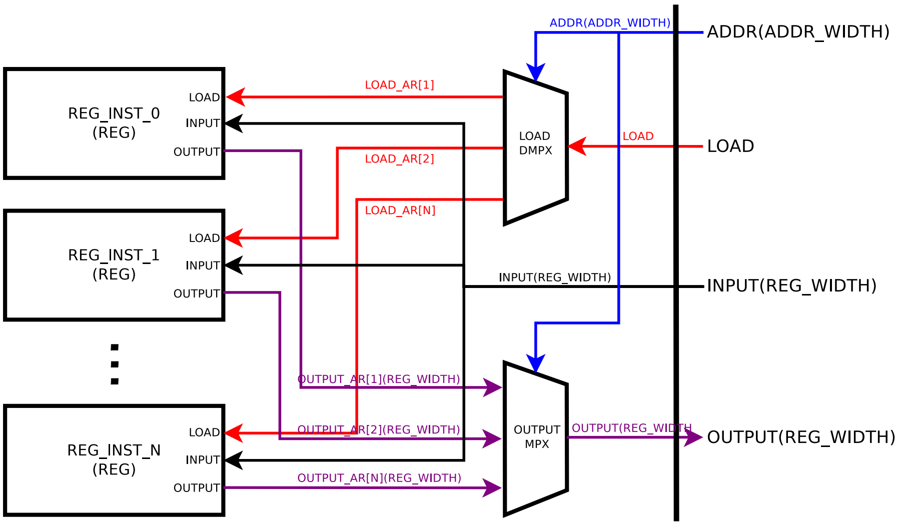

Figure 3.10: Block diagram of REG_CONCAT's datapath.

entities. *OUTPUT_MPX* is implemented as an array-type element access for the same reason.

MEM is instantiated as *INPUT_MEM*, *INPUT_MEM_CONST* and *OUTPUT_MEM*. *ADDR_HIGH* addresses individual *REG_CONCAT* instances inside the memory, and it is always connected to controller's *REG_CNT_VAL*, counting over all inputs or outputs. *ADDR_LOW* addresses *REG* instances inside *REG_CONCAT* instances, and it is connected to controller's *SHARES_CNT_VAL*, counting over all shares of an input or an output.

Generics		
REG_CNT		number of inputs/outputs to be stored in the memory
REG_SHARES_CNT		array of naturals; number of shares of each input/output
REG_WIDTHS		array of naturals; number of bits of each input/output
REG_MAX_WIDTH		maximum of REG_WIDTHS
ADDR_HIGH_WIDTH		width of ADDR_HIGH signal
ADDR_LOW_WIDTH		width of ADDR_LOW signal
Ports		
CLK	in	clock input
RESET	in	reset input
INPUT	in	data to be saved to addressed register
LOAD	in	when '1', save INPUT to addressed register
ADDR_HIGH	in	address signal; input/output choosing
ADDR_LOW	out	address signal; share choosing

Table 3.12: MEM entity interface.

The advantage of using this hierarchical memory description instead of a 2D array of bit-vectors (with, e.g., INPUT_MEM(REG_CNT_VAL)(SHARES_CNT_VAL) element access in VHDL) is an easier optimalization during synthesization process. For example, the *OUTPUT_AR[0]* signal (violet in Figure 3.11) is connected to *REG_WIDTHS[0]* bottom bits of *REG_CONCAT_INST_0's OUTPUT* port, and the rest is assigned to logical zero, which helps the synthesizer to remove unnecessary bits from the final FPGA design. The synthesizer cannot foresee which bits will be useless at runtime when the array construct is used.

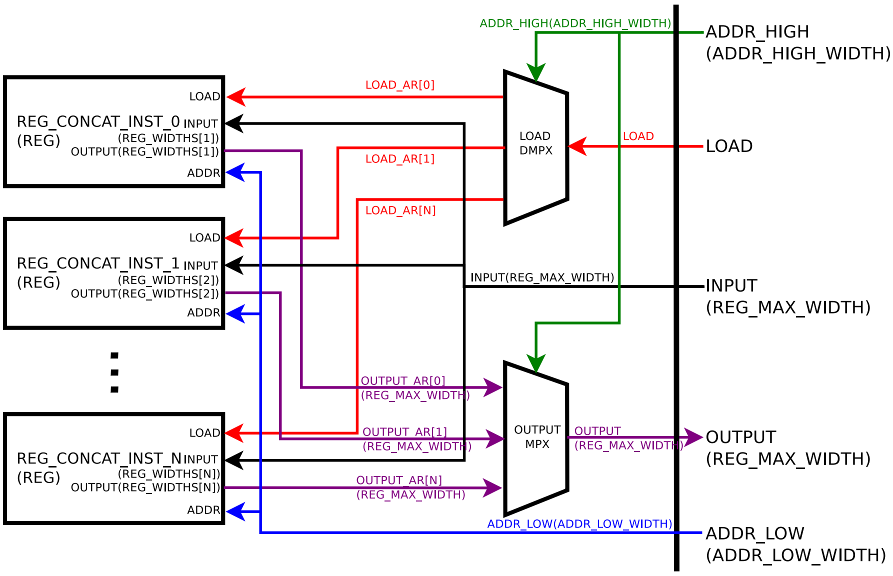

Figure 3.11: Block diagram of MEM's datapath.

MASKER implements the data-remasking logic. Its interface is in Table 3.13.

Generics		
SHARES_CNT	maximum number of shares to be expected	
REG_WIDTH	maximum input width to be expected	
ADDR_WIDTH	width of the address signal	
Ports		
CLK	in	clock input
RESET	in	reset input
INPUT	in	data/mask input
LOAD_DATA	in	when '1', save INPUT to the addressed data register
LOAD_MASK	in	when '1', save INPUT to the addressed mask register
REMASK	in	when '1', create new remasked data from the internal register values
ADDR	in	address signal
OUTPUT	out	addressed data register output

Table 3.13: MASKER entity interface.

The entity is also depicted in Figure 3.12.

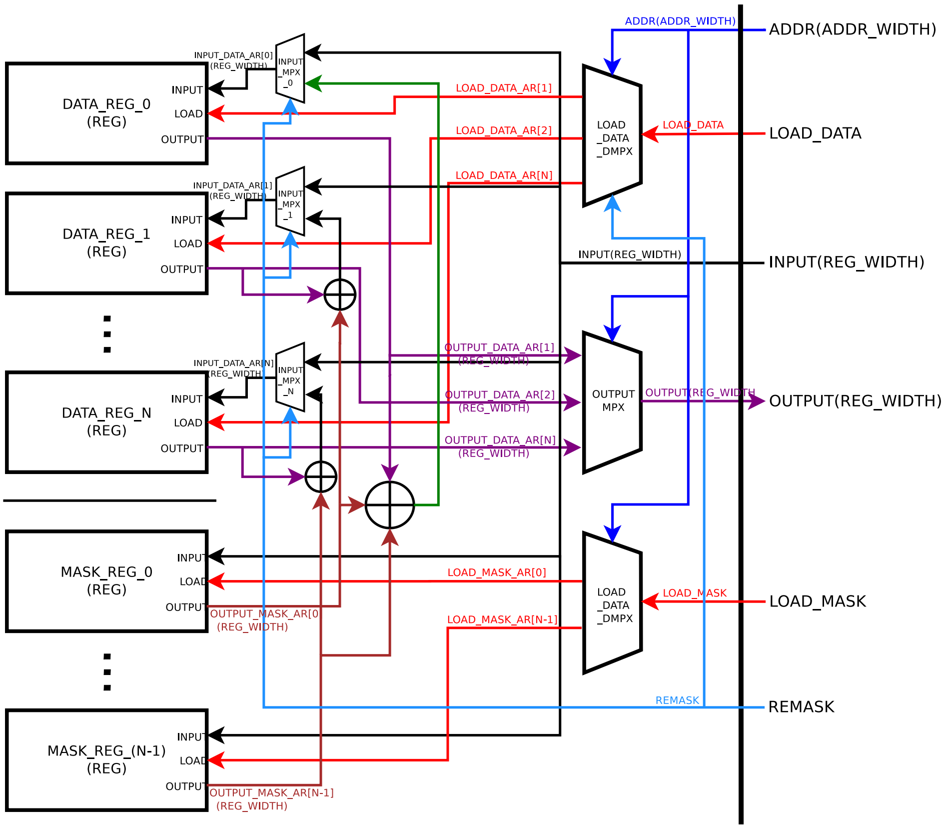

Figure 3.12: Block diagram of MASKER's datapath.

Shared data are expected to be in the following form: $(masked\_data, m_0, m_1, ..., m_n) = (orig\_data \oplus m_0 \oplus m_1 \oplus ... \oplus m_n, m_0, m_1, ..., m_n)$. The multiplexing and demultiplexing blocks in Figure 3.12 are created as processes in VHDL. *LOAD_DATA_DMPX* works as a demultiplexer when *REMASK* value is logical zero, otherwise all *LOAD_DATA_AR* elements are logical one. The number of mask registers is one less than the number of data registers, since the first data register contains the masked data. Remasking occurs in the following way:

- All of the mask register's outputs are exclusive-ored (XORed) together and the resulting value is XORed to the first data register's value and saved back into the register.
- Mask register $0..(N-1)$ outputs (new masks) are XORed to data register $1..N$ value, respectively. The resulting value is saved into the respective data register.

This way the unmasked value never appears anywhere in the design.

3.1.2.2 Control's Controller

The Control's FPGA controller is realized as a large Moore FSM decomposed into sub-automaton for better readability and maintainability. It also contains four *COUNTER* instances and one *PRNG* instance:

- **N_COUNTER** - counts down the number of encryptions remaining.
- **REG_COUNTER** - used to loop over all inputs/outputs. Looping over all inputs/outputs is denoted in the FSM diagrams in red color.
- **SHARES_COUNTER** - used to loop over all shares of a particular input/output. Looping over all shares of an input/output is denoted in the FSM diagrams in blue color.
- **TIMER** - this timer ensures that the duration between encryptions is at least the specified number of clock cycles. For example, the Picoscope oscilloscope requires 1 μs re-arm time between individual trigger signals [9].
- **PRNG_INST** - PRNG used to decide between constant and random input in Fixed vs. random operation mode.

The description of individual automaton follows.

MAIN_FSM is the top-level entity of automaton decomposion hierarchy. It starts individual sub-automaton using *[sub_automaton_name]_START* signal and waits for its runtime end confirmed by the *[sub_automaton_name]_END* signal. It also implements the main loop of Control's operation — each loop realizes one data preparation–encryption–output data handling sequence. The main loop is denoted in green color in figure 3.13.

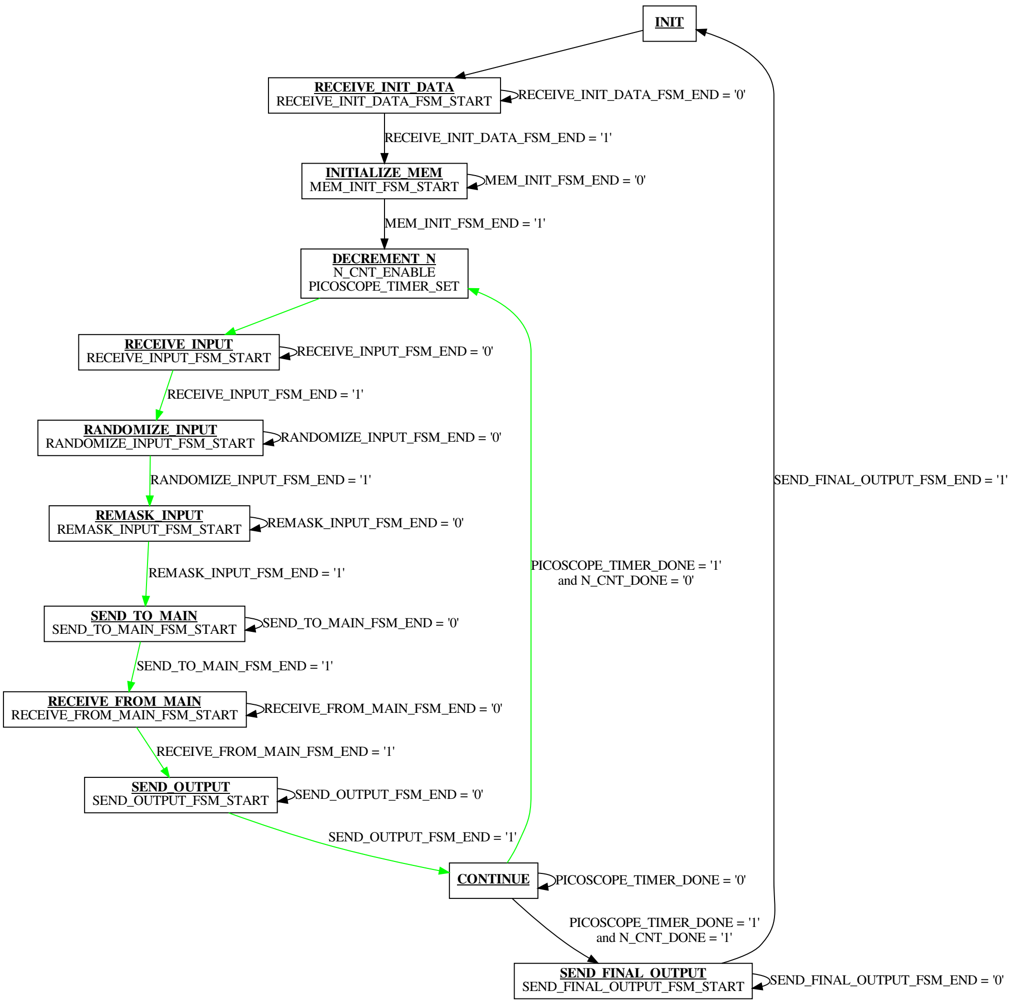

Figure 3.13: Main FSM of Control's controller diagram.

RECEIVE_INIT_DATA FSM controls the initial data receive sequence. The lower layers of receiving are implemented in the *UART_WRAPPER* instantiation in Control. This automaton just tells it how many bytes of data shall be received in the *WAIT_FOR_RS232*, *WAIT_FOR_RS232_2*, *RECEIVE_N*, *RECEIVE_DATA_SEED* and *RECEIVE_CONTROL_SEED* states. First, the input and output operation codes are received for every input and output separately. Then, number of encryptions to be done is received, and the *N_CNT* counter in Control's controller is initialized with the received value. The data seed follows and it is used to initialize *DATA_PRNG*. Lastly, the controller seed is received and it used to initialize *CONTROL_PRNG* in Control's controller.

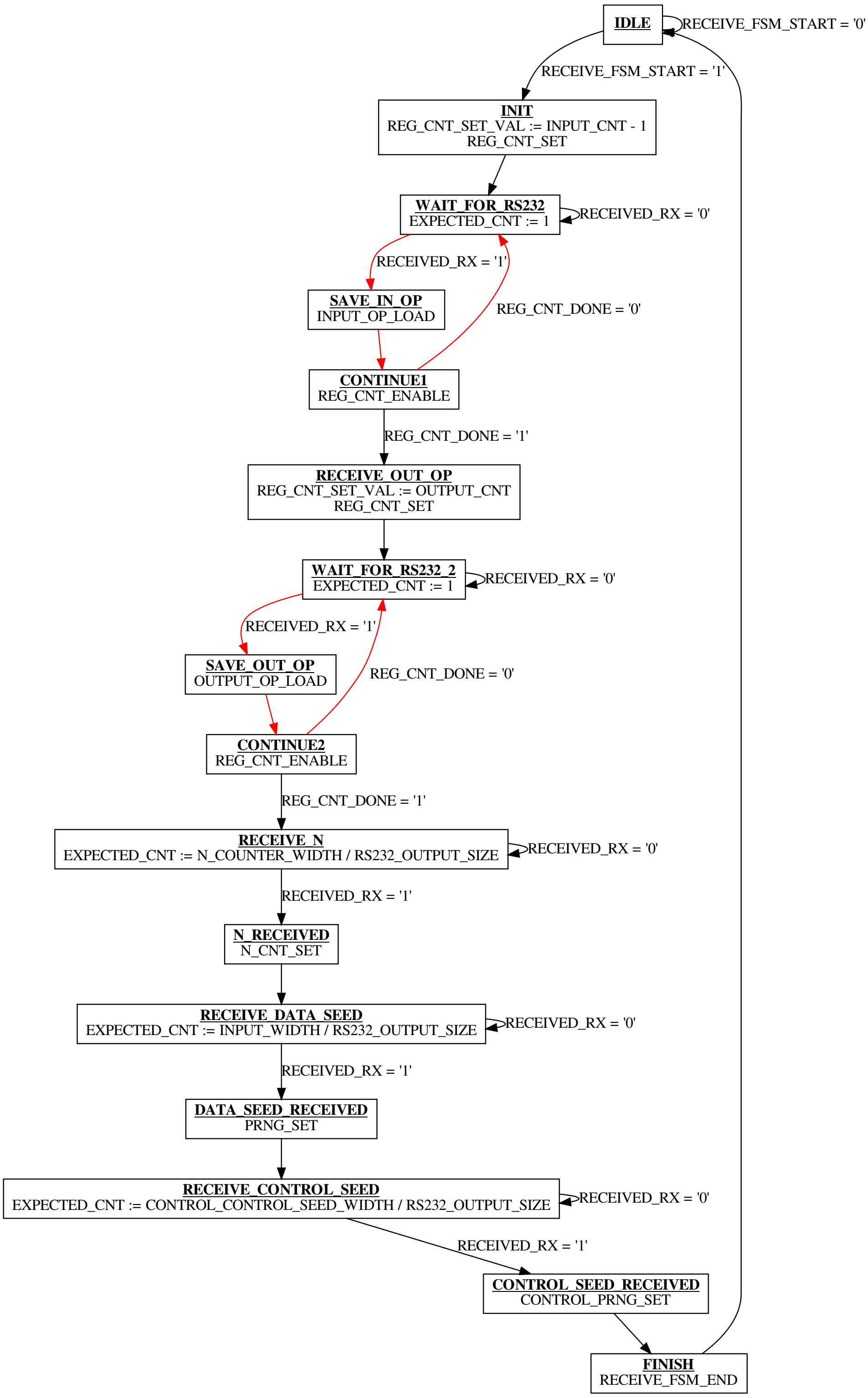

Figure 3.14: Receive init data FSM diagram.

MEM_INIT FSM loops over every input and every share of the input to fill it with either data from the computer (the left blue loop) or *DATA_PRNG* output (right blue loop) based on the input operation code value. Both *INPUT_MEM* and *INPUT_MEM_CONST* instances are filled with the same received or generated data. The lower layers of receiving are implemented in the *UART_WRAPPER* instantiation in Control. This automaton just tells it how many bytes of data shall be received in the *RECEIVE* state.

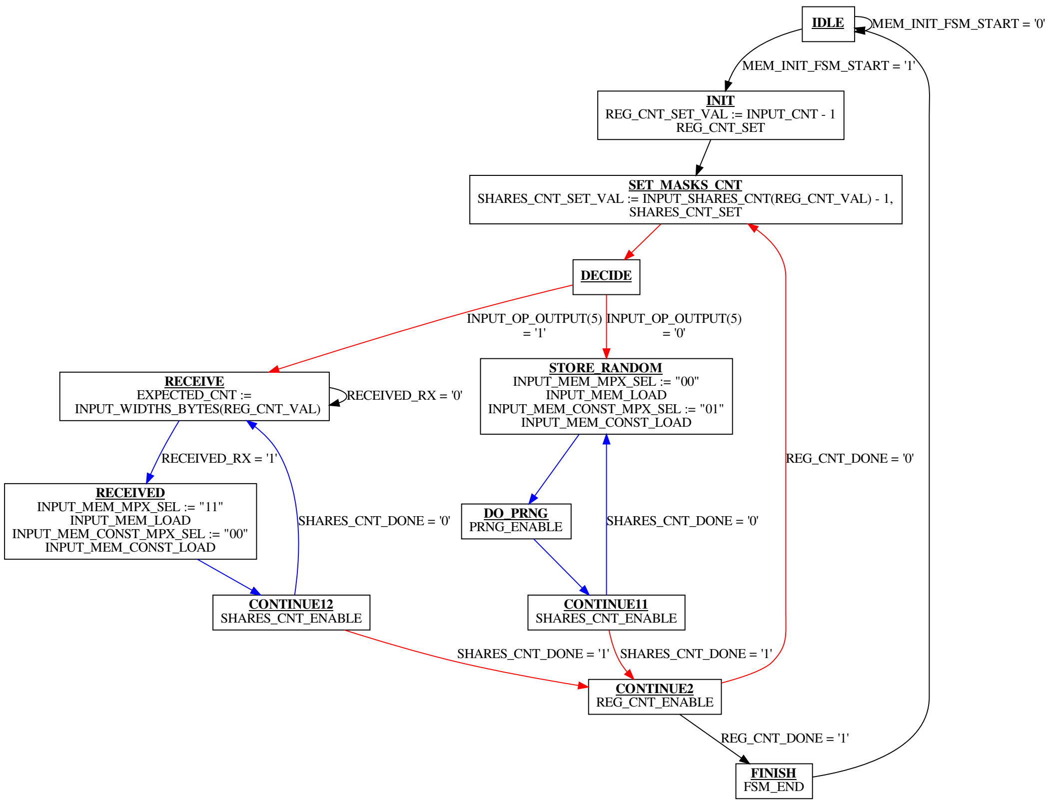

Figure 3.15: Initialize memory FSM diagram.

RECEIVE_INPUT FSM receives input data from the computer, just like one branch of the Initialize Memory FSM. The difference is that this automaton fills only the *INPUT_MEM*, whereas Initialize Memory FSM fills both *INPUT_MEM* and *INPUT_MEM_CONST*, and that this automaton is run in the main loop — the data are received before every encryption.

DECIDE state starts or skips data receiving based on the input operation code. The lower layers of receiving are implemented in the *UART_WRAPPER* instantiation in Control. This automaton just tells it how many bytes of data shall be received in the *RECEIVE* state. Shall the receiving of a particular input happen, one byte of random data is sent to the computer for synchronization purposes – otherwise the computer could start data sending too soon and some data might get lost since there is no buffer in the *UART* entity. The input data are then received in the share-by-share manner.

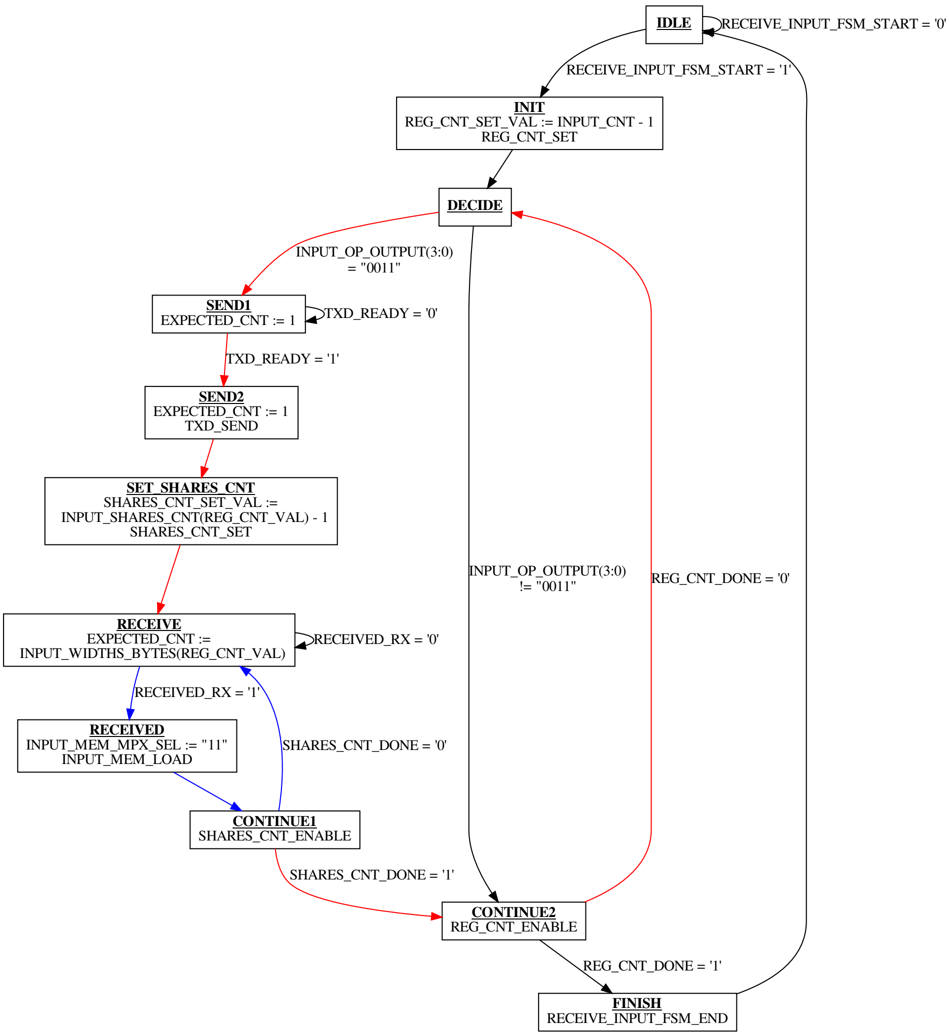

Figure 3.16: Receive input data FSM diagram.

RANDOMIZE_INPUT FSM loops over all inputs of *INPUT_MEM* and XORs *DATA_PRNG* output to the first share of the input. Since the other shares always contain individual masks, no randomization shall occur there.

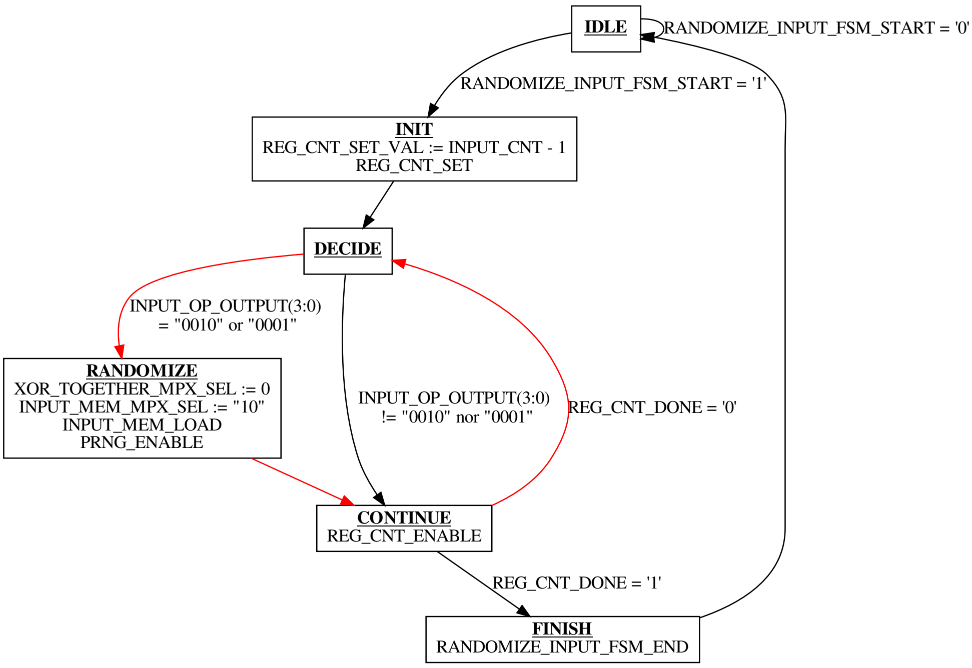

Figure 3.17: Randomize input data FSM diagram.

REMASK_INPUT FSM does remasking of the input data. *DECIDE* state starts or skips data remasking based on the input operation code. *MASKER_RESET* signal is active during *DECIDE* state. It is connected to the reset input of *MASKER* entity and thus zeroes its register's values between individual remasking runs. The remasking is also naturally skipped when the current input data have only one share. Then, the *INPUT_MEM* data (left upper blue loop) or *INPUT_MEM_CONST* data (right upper blue loop) are loaded into the *MASKER* entity. Constant data are remasked in fixed or Fixed vs. random mode when the next input data shall be fixed. After that, new masks are loaded into the *MASKER* from *DATA_PRNG* entity (cyan loop). Next, the data are actually remasked inside of the *MASKER* entity without the unmasked input value appearing anywhere in the design. Finally, remasked data are moved back into *INPUT_MEM* (left bottom blue loop) or *INPUT_MEM_CONST* (right bottom blue loop) based on the operation mode.

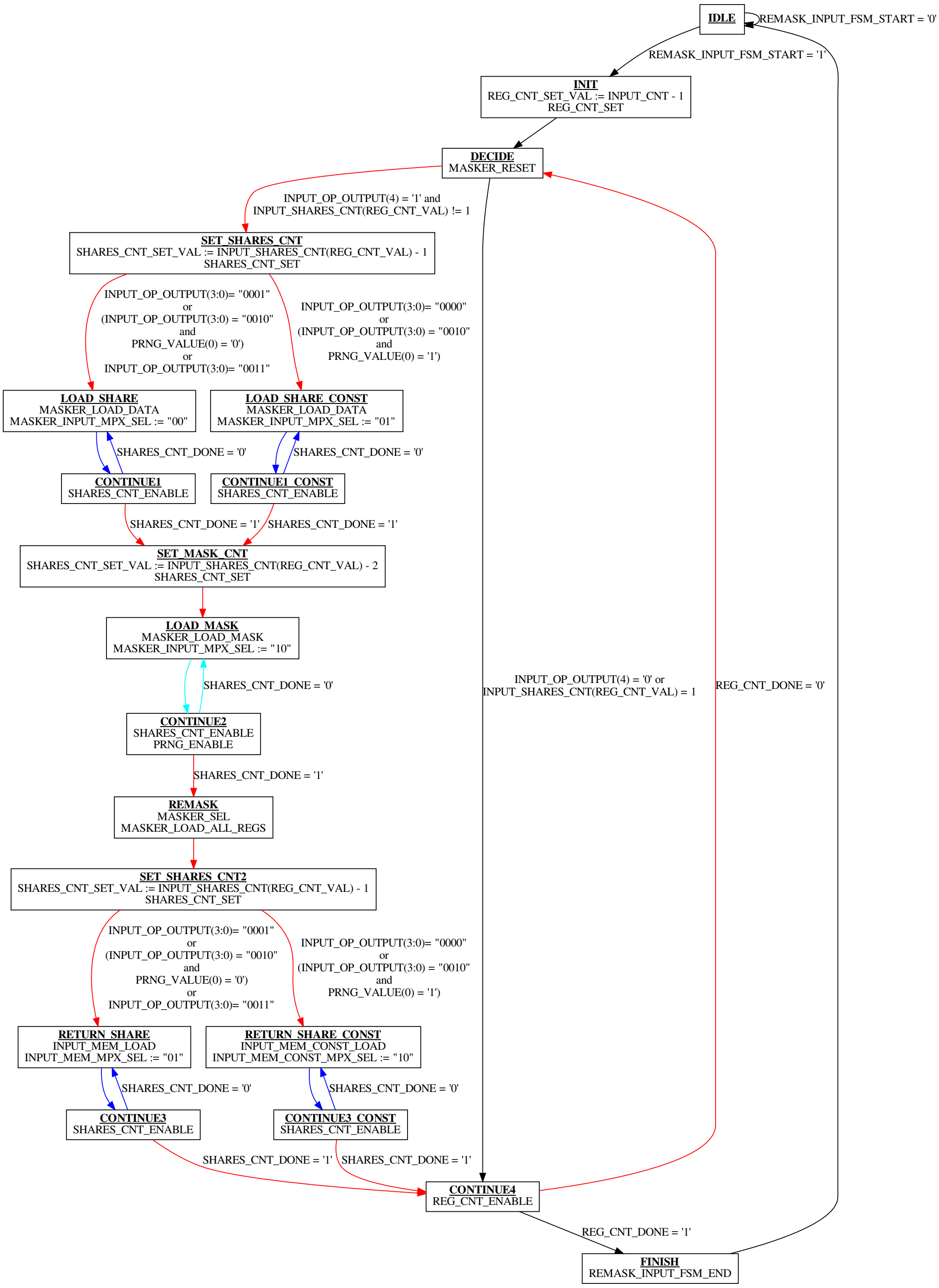

Figure 3.18: Remask input FSM diagram.

SEND_TO_MAIN FSM sends input data to Main. The lower layers of sending are realized inside of the *INTER_FPGA_COMM* entity instance in Control. This automaton just tells it how many *INTER_FPGA_WORDS* to send and how many to skip in the *SEND* state.

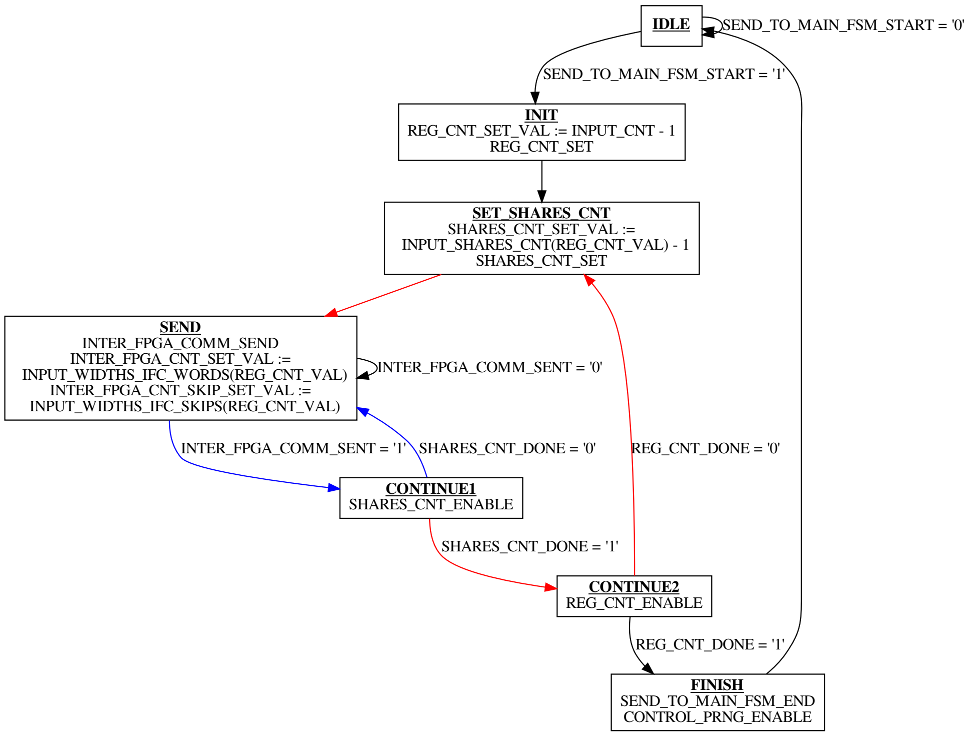

Figure 3.19: Send to main FSM diagram.

RECEIVE_FROM_MAIN FSM receives input data from Main. The lower layers of receiving are realized inside of the *INTER_FPGA_COMM* entity instance in Control. This automaton just tells it how many *INTER_FPGA_WORDS* to receive in the *RECEIVE* state.

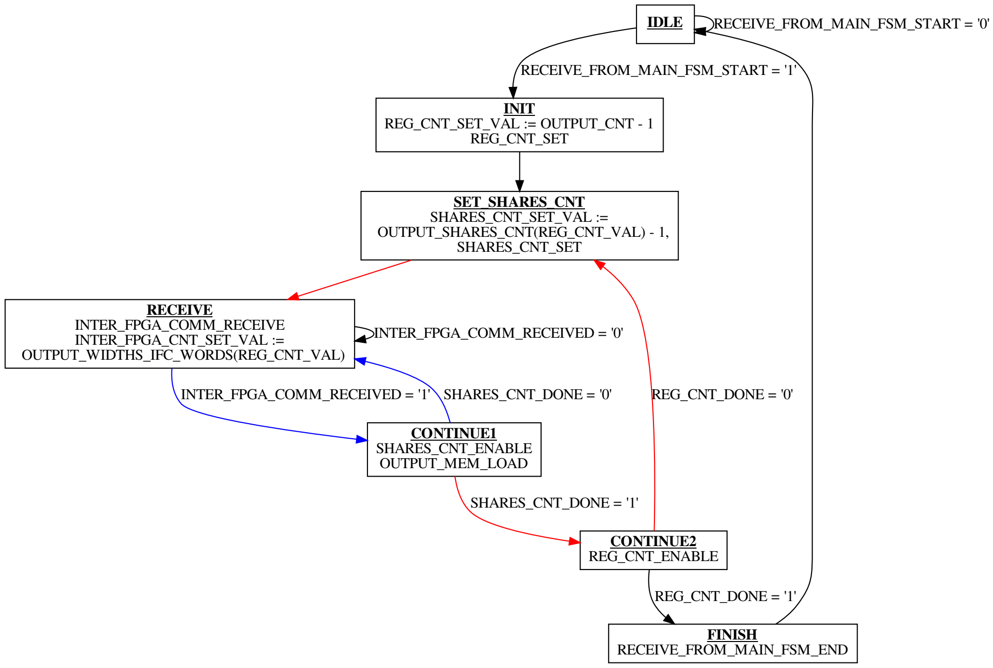

Figure 3.20: Receive from main FSM diagram.

SEND_OUTPUT FSM sends cipher's output data to the computer. Whether the current output shall be sent or not is decided in *DECIDE* state based on the output operand. The lower layers of sending are implemented in the *UART_WRAPPER* instance in Control. This FSM just tells it how many bytes to send and how many bytes to skip in the *SEND2* state.

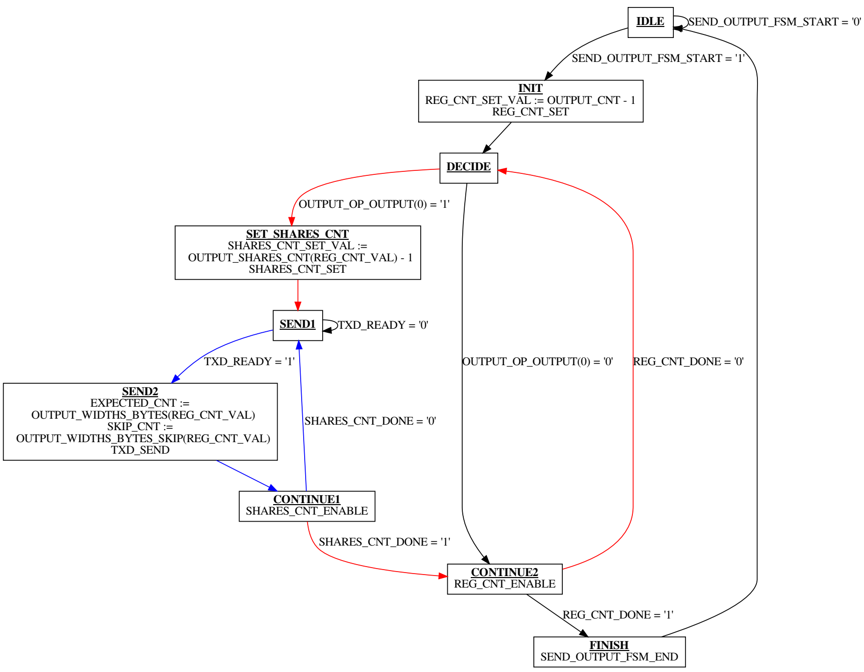

Figure 3.21: Send output FSM diagram.

3.1.3 Main

The design created for Main FPGA serves as a wrapper for the cipher. The wrapper's main functionality is the following procedure:

- Receive data from Control.
- Put the received data on cipher's input and start the encryption.
- Upon encryption's end, save the cipher's output.
- Send the saved data back to Control.

Additionaly, the wrapper has the following functions:

- **Trigger** - set the trigger signal on in two different ways:
  - On for one clock cycle before pre-delay.
  - On during the whole pre-delay, encryption and post-delay.
- **Pre-delay** - Upon receiving the data, wait for specified number of clock cycles before launching the cipher.
- **Post-delay** - Upon the end of encryption, wait for specified number of clock cycles before output data send starts.
- **Random cipher input** - Create random data for the optional random cipher input of the specified width.

Main's interface is in Table 3.14.

Ports		
CLK_ORIG	in	clock input; it is assigned directly to CLK signal, which is used as the clock source for sub-entities; optional clock divisor can be inserted between CLK_ORIG and CLK
RESET	in	reset input
PT_REQ	in	data receive from Control request
CT_ACK	in	data send to Control acknowledge
PT_PART	in	data bus from Control
PT_ACK	out	data receive from Control acknowledge
CT_REQ	out	data send to Control request
CT_PART	out	data bus to Control

Table 3.14: TOP_MAIN entity interface.

The cipher shall be instantiated in Main. *INPUT_DATA_REGISTERED* is an two-dimensional array of *std_logic_vector(WIDTH - 1 downto 0)*, where

WIDTH is the maximal width of all the inputs and all the outputs. The inputs and its shares are indexed in the same manner as in the *config.txt* file. If the input to be connected is smaller than *WIDTH*, it must be connected to lower bits, e.g., third share of the second input with the width of 80 bits will be connected to *INPUT_DATA_REGISTERED(1)(2)(79 downto 0)*. The outputs are connected to the *OUTPUT_DATA_CIPHER* signal in the same manner. The signal which starts the encryption by a high pulse for one clock cycle is the *START_ENCRYPTION*. The signal indicating that the encryption has finished by its high value is the *ENCRYPTION_DONE*.

3.1.3.1 Main's Datapath

Main's datapath is depicted in Figure 3.22.

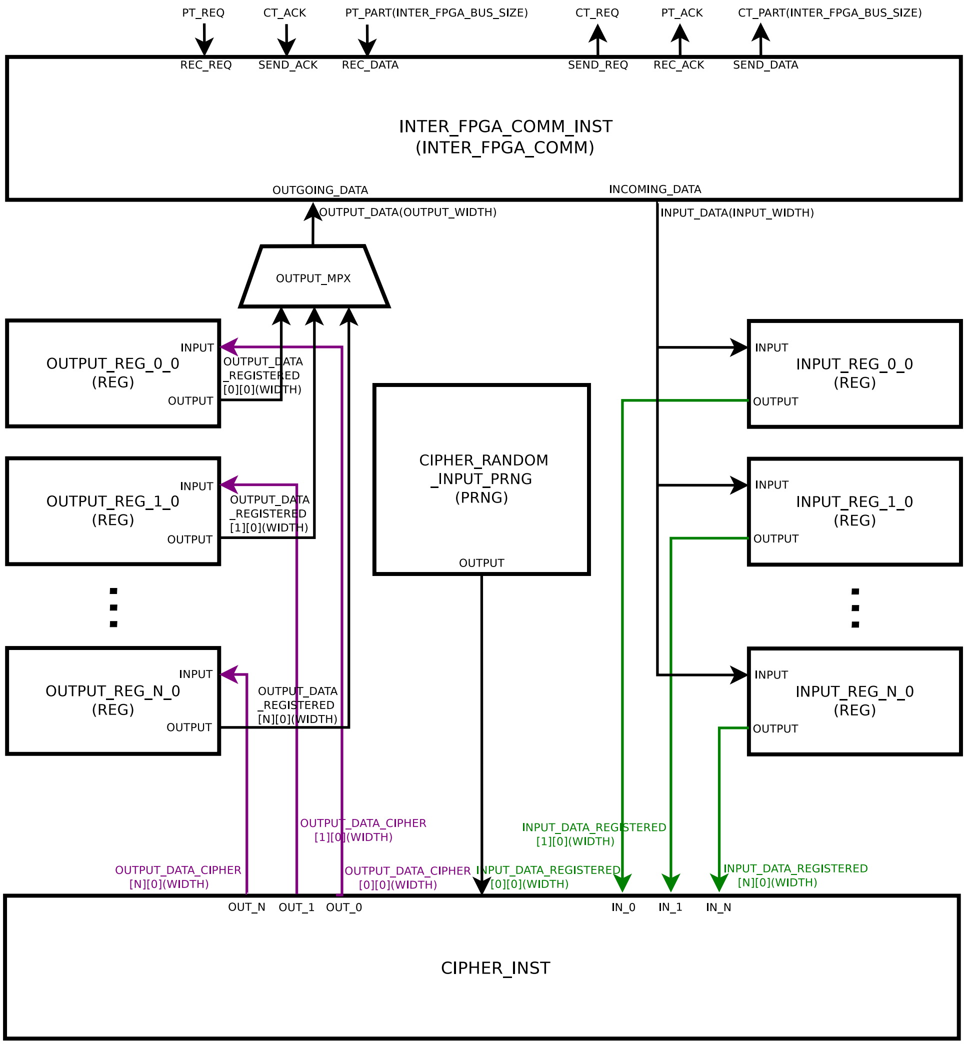

Figure 3.22: Block diagram of Main's datapath. Only one share per input/output is considered for better readibility.

No *MEM* entity is instantiated here, because all of the received inputs have to be accessible at once. Similarly, the output registers have to be loaded with the data from the cipher at once. Therefore, nested VHDL's for-generate construct is used to generate as many *REG* instances as required. Input registers' loading is controlled by a load signal of array type, similar to these used in *REG_CONCAT*, *MEM* and *MASKER* entities. *OUTPUT_MPX* is implemented as an array element access in VHDL. *PRNG* instance has *PRNG_ENABLE* signal connected to logical one and thus its output value changes every clock cycle.

3.1.3.2 Main's Controller

Main's controller is realized as one Moore's FSM. It also contains four *COUNTER* instances:

- **REG_COUNTER** - used for looping over all inputs or outputs. Looping over all inputs or outputs is denoted in the FSM diagram in red color.

- **SHARES_COUNTER** - used for looping over all shares of a particular input or output. Looping over all shares of an input or output is denoted in the FSM diagram in blue color.

- **PRE_COUNTER** - used for counting of the pre-delay duration.

- **POST_COUNTER** - used for counting of the post-delay duration.

A state diagram of the controller's FSM is in Figure 3.23. First, input data are received from Main. Lower layers of the communication are implemented in *INTER_FPGA_COMM* instance — here the automaton just declares how many *INTER_FPGA_WORDS* to receive in *RECEIVE* state. Trigger set follows, then the pre-delay and finally the encryption itself. After the encryption is done, the post-delay and the final output data sending to Control follows. The number of *INTER_FPGA_WORDS* to send and to skip is declared in the *SEND* state.

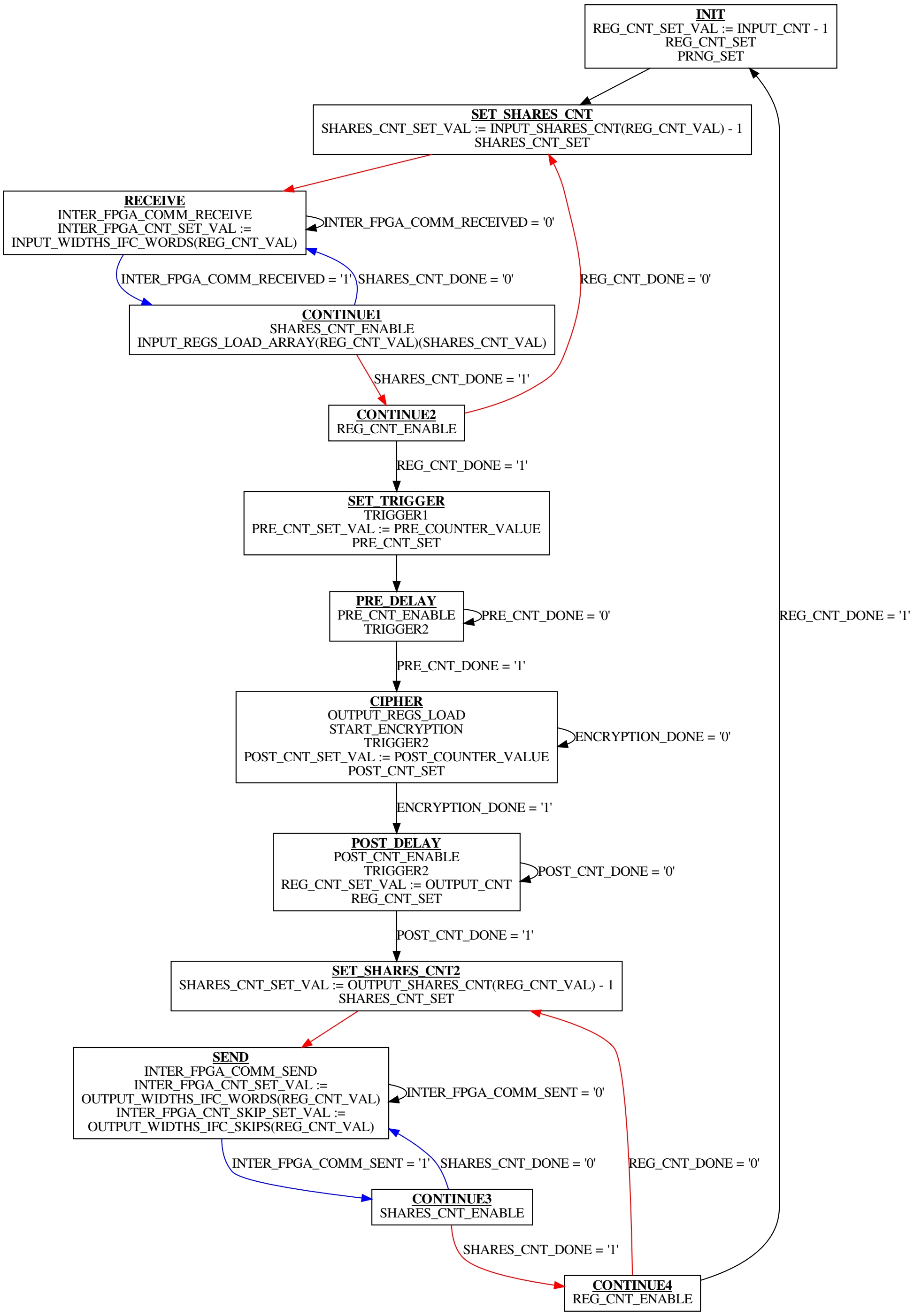

Figure 3.23: Main controller's FSM state diagram.

3.2 Verification

Unsynthesizable models of both Control and Main were created in VHDL for verification purposes. In this section, I call the unsynthesizable description the "model" and the synthesizable description the "RTL".

The top testbench (TB.vhd) instantiates both the model and RTL of Control and Main. The same sequence of the input data is sent to the model and RTL of Control. The correctness of the run is then checked in the following manner:

- Inter-FPGA communication of the model and RTL is compared.
- Received output data are saved and, at the very end of the testbench's runtime, model's and RTL's output data are compared.

Since there are no real-time delays in the model, such as waits for clock's rising edge, the model is naturally faster than RTL. Synchronization of model's and RTL's runtime happens at the inter-FPGA communication point. The TB entity waits for the request signal from both RTL and model, then it compares the inter-FPGA data bus and only after that the request signal is passed over to its destination. The TB continues in a similar way with acknowledge signal. Described process is depicted in Figure 3.24.

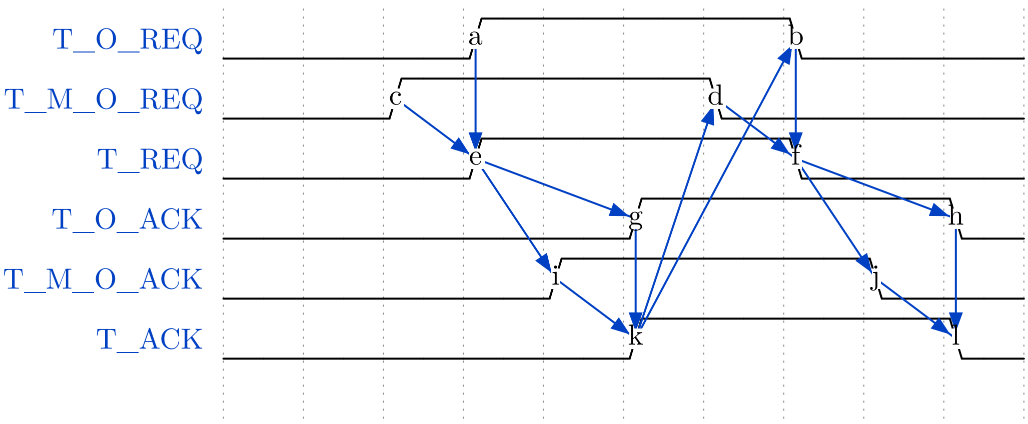

Figure 3.24: Testbench inter-FPGA communication synchronization. Signals prepositioned with T_O_are RTL's output. Signals prepositioned with T_M_O_ are model's output. Other signals are the RTL's and model's common input driven by the main process of TB. Arrows describe order of the changes of the signals.

Ideas of constrained-random verification are included in the testbench – it creates a random input based on the *SEED1* and *SEED2* variables in *DEFIN-*

ITIONS package using the Uniform function of IEEE_MATH_REAL package. The input and output operation codes are created randomly with the constrain that the resulting code must be a valid one. Random input data are created without constrains.

At first, the design was simulated using the Modelsim 10.4a PE Student Edition [7], but the runtime was unsatisfactorily long due to the design's size exceeding student's edition recommended capacity. After that, the open-source VHDL simulator GHDL [13] was used. Not only did it run faster, but, unlike Modelsim, GHDL also allows more instances to be run at once, effectively allowing easy parallelization of the constrained-random verification.

To verify the design with as many configurations as possible, and to make the verification more convenient, a script called *verify.py* written in Python 3 language was created. The script's options are shown in Table 3.15. The options are also printed to the standard output when the script is launched without parameters. More options, for example maximum width of an input in the randomly generated configuration or some of the default values, can be set at the beginning of the script file in the "editable" section. Simulator's output is not shown on the standard output of the script and it is saved to log.txt file instead.

Switch	Default value	Description
-m	none	Mode of operation: random - generate DEFINITIONS.vhd randomly; fixed - generate DEFINITIONS.vhd using config.txt file.
-g	none	When present, only generate DEFINITIONS.vhd file and do not run simulator.
-r	1	Number of simulator runs to be launched.
-e	GHDL	Simulator to be used: GHDL or MODELSIM. GHDL uses source.lst as a list of files to be analyzed and elaborated. Modelsim uses run_cmd.do.
-t	1800	Simulator runtime timeout in seconds.
-s	random	Simulator run seed value. When an error is discovered, the seed is saved to errlog.txt and the same run can be repeated using this option.
-w	none	When present, run the simulator's GUI. Only works for Modelsim and it uses run.do as source list file instead.
-p	1	Number of threads to be launched. Every thread launches its simulator instance. Shall be less or equal to number of repetitions. For Modelsim it is ignored.
-n	10	Number of encryptions to be simulated.

Table 3.15: verify.py script options.

Chapter **4**

SICAK Plugin

A measurement scenario plugin for Software Toolkit for Side Channel Attacks (SICAK) [11] was created. The plugin is called *sakurag* and it serves as the software counterpart of the FPGA design described in Section 3.1. The plugin is run using the *meas* utility of SICAK. More info about its functionalities and parameters can be found in [11]. The *sakurag* specific options are passed to the plugin using the *param* option. The string passed after the *param* option shall consist of key=value pairs separated by semicolons. The accepted key=value pairs are in Table 4.1.

The list of *sakurag* class method follows. The inherited methods described in [11] were skipped.

parseParams parses the input/output parameters stripped of the in/out preposition, respectively.

loadConstantInputs loads initialization data from the JSON configuration file.

lfsr32 implements one shift of *uint32_t* data using the [32, 22, 2, 1] LFSR register [4].

doDataPrngStep does one shift of the *m_dataPrngValue* using parallel LFSRs just like the *DATA_PRNG* of the Control does.

doControlPrngStep does one shift of the *m_controlPrngValue* using the lfsr32 function just like the *CONTROL_PRNG* of the Control's controller does.

sendInitData sends the initialization data received by the *RECEIVE_INIT_DATA* FSM of the Control's controller.

sendMemInit sends the input initialization data received by the *MEM_INIT* automaton of the Control's controller. When the data are generated randomly inside of the Control instead, the method generates the same data and saves them to *m_inputMem* and *m_inputMemConst*.

sendInput sends the input data received by the *RECEIVE_INPUT* FSM of the Control's controller. The data are generated randomly.

randomizeInput randomizes the input data saved in the *m_inputMem* variable in the same way that the *RANDOMIZE_INPUT* FSM of the Control's controlles does.

remaskInput remasks the input data in the same way that the *REMASK_INPUT* FSM of the Control's controlles does.

receiveOutput receives the output data sent by the *SEND_OUTPUT* automaton of the Control's controller.

send serves as the *chardevice* instance's send method wrapper. The data shall be sent MSB to LSB, therefore this method calls the *chardevice's* send byte-by-byte in the correct order.

createInputs unmasks data saved in the *m_inputMem* or *m_inputMemConst* class variable and saves it into the *m_inputs* variable. The unmasked data are later used for file saving.

createOutputs unmasks data saved in the *m_outputMem* class variable and saves it into the *m_outputs* variable. The unmasked data are later used for file saving.

stripExtraBits the plugin works internally with byte values, but the device support any width value. The extra bits, which might not be empty due to the *m_dataPrngValue* being XORed onto it (the *m_dataPrngValue* is always 32*k bits wide), are stripped away in this method.

runCipher serves as a wrapper for the cipher method call. Method implementing the cipher instantiated in Main shall be called in this method. Depending on the cipher method, masked or unmasked data input will be required. The masked input data are passed to this method in the *currentMem* parameter — it is a pointer to either the *m_inputMem* or the *m_inputMemConst* depending on the mode of the input. The unmasked data are available in the *m_inputs* class variable – it is already filled with the correct data, random or constant, when this method is called.

testMode is called when the test param is on. Both the output value precomputation and the receiving from the device take place in the test mode. The data are then compared and the unequalities are reported. To compute the cipher's outputs, a software implementation of the cipher instantiated in the FPGA design is required. The *dummyCipher* method can be used as the cipher in software — *DUMMY_CIPHER* entity shall be then instantiated in Main. No data saving or communication with the oscilloscope takes place in the test mode.

compareMems is called by the *testMode* to compare the received and computed output values.

dummyCipher can accomodate to any number of inputs and outputs and it always places the first share of the first input onto all outputs. Only the fitting part of the input is used when the input and output widths are different. Its hardware implementation is in the *DUMMY_CIPHER* entity.

allocSpace allocates the required space for the class variables of *SakuraG* class. It is called at the beginning of the *run* method, since the number of measurements is required for the memory allocation.

deallocSpace de-allocates the space allocated in allocSpace. It is called at the end of the *run* method.

Internally, the plugin does the same computations as the FPGA device. At first, the plugin creates operation codes, random data and control seed based on the received parameters in the *init* and *parseParams* methods and sends it together with the number of encryptions to the device using the *sendInitData* method. Shall any input be initialized from the computer, the *sendMemInit* method sends the data loaded from the JSON configuration file. Otherwise, it generates the same initialization values that the device does. The main loop (green in Figure 3.13) follows. Shall any input receive data from the computer before every encryption, the *sendInput* method sends randomly generated data. Appropriate inputs in *m_inputMem* are randomized in the *randomizeInput* method. Remasking occurs in the *remaskInput* method. Extra bits are stripped off the inputs in the *stripExtraBits* method. Cipher's input data are unmasked and saved into the *m_inputs* class variable in the *createInputs* method. These data are later used for the file saving. If the cipher software implementation is available, it is run in the *runCipher* method to generate the outputs. The outputs meant to be received after every encryption are received in the *receiveOutput* method.

The JSON file with the initialization input values shall contain key : value pairs, where key shall be *inputI* (I is the input index) and value shall be a hexadecimal string with the appropriate length. The length shall be nibble-accurate, e.g., an input of a width of ten bits shall be described by a string of three hexadecimal digits.

The plugin saves the inputs, the outputs and the traces into files. Every input and every output is saved to a separate input/output file, respectively. Inputs and outputs are saved in the unshared form. The structure of input/output file name is {in,out}put{I,J}-measurementID.bin, where I is the index of the input; measurementID can be set using meas' *id* param and it is the current datetime by default. Traces are saved to {random,constant}-traces-measurementID.bin file. The constant traces file is selected either when all of the inputs use Fixed mode or when one of the cipher's inputs uses Fixed vs. random mode of operation and the cipher's input was fixed during the trace's measurement. When the constant traces file is used, no inputs/outputs are saved, thus the input/output inside of the respective files are alligned to the traces inside of the random traces file.

Switch	Default value	Description
inI=N	none	Width of the cipher's input port I is N bits. I shall start at zero and no index shall be skipped.
outJ=N	none	Width of the cipher's output port J is N bits. J shall start at zero and no index shall be skipped.
inshareI=N	1	The cipher's input port I consists of N shares.
inshareJ=N	1	The cipher's output port J consists of N shares.
inmodeI=S	random	Mode of the the cipher's input port I is S, where S shall be either of: • fixed • random • randomvsfixed (or fixedvsrandom) • fromPC See Section 2.3 for the description of the listed modes.
outmodeJ=S	nosend	Mode of the the cipher's output port I is S, where S shall be either of: • nosend • toPC See Section 2.3 for the description of the listed modes.
inremaskI=N	1	Remasking of the cipher's input I is on (N=1) or off (N=0).
ininitPCI=N	0	Cipher's input I is initialized from the computer (N=1) or using inner PRNG (N=0).
inconstfile=S	empty string	JSON configuration file containing initialization input values.
test=N	0	Run the test mode (N=1).

Table 4.1: Accepted key=value pairs in the string passed as the *param* parameter of the SICAK's *meas* utility.

CHAPTER 5

System Integration and Testing

In this chapter, system integration and the following testing is described.

5.1 System Integration

At first, the Control's and Main's design had to be synthesized. Xilinx ISE [15] was used to create the programming file. After that, Sakura-G's FPGAs had to be programmed. Digilent USB-JTAG Programming Cable together with the Impact software (part of the tools bundled with ISE) was used to program the FPGAs. Various Control's controller FSM states were encoded into binary form and the resulting number was connected to Sakura-G's built-in LEDs to see what was happening in the device. Sakura-G provides FTDI FT2232H [3] chip for communication over USB – the communication with the device was checked using Bash's Echo built-in function redirected to the serial device. When the manual communication using the Bash seemed to work correctly, it was time to test the SICAK plugin. To make the first steps of the integration of the FPGA device with the plugin easier, a simple configuration consisting of one input/one output was loaded into the FPGA. Then, SICAK plugin was run with the appropriate serial device parameter and *param* string. The plugin was also modified to send zero data-seed into the device for easier navigation through the received outputs. The bugs of the plugin, discovered by observing the LED outputs of Sakura-G and by a thinking about the inner working of the design, were removed one-by-one until one full run of the device was achieved. Finally, different modes of the inputs were tested and eventually fixed in the same manner.

5.2 Testing

To test the integrated system with as many different configurations as possible, a script called *test.py* written in Python3 language was created. The script automates the testing process by using the *DEFINITIONS.vhd* generator from *verify.py* to generate the file randomly/using the *config.txt* file; by running the synthesize-translate-map-place&route-generate programming file toolchain of Xilinx ISE to generate the programming file for both Control and Main; by downloading the programming file to both FPGAs using Xilinx Impact; and finally by running the SICAK *meas* with sakurag plugin in the test mode repeatedly with all the possible configurations for every cipher's input. The dummy-cipher described in Section 4 was used. The script's options are shown in Table 5.1. The options are also printed to the standard output when the script is launched without parameters. Paths to Xilinx tools, to the SICAK installation directory, the communication port, and the communication port config file shall be set at the beginning of the script file in the "editable" section. Default values of the script's parameters can be adjusted at the same place. *DEFINITIONS.vhd* file random generation constraints, like the maximum number of shares, can be adjusted at the beginning of the *verify.py* script.

Since there were no software tests created for the SICAK plugin, its functionality is verified solely by the described process. It shall suffice, because:

- The FPGA design was verified using the testbench environment with unsynthesizable models as described in Section 3.2.
- The plugin was tested against the FPGA design as described in this section.

Therefore, three independent descriptions of the device were created and their functionalities were compared. At least 10000 simulator runs, each with different configuration, were executed to thoroughly verify the design. At least 300 different configurations were uploaded into the device and they were tested using various input/output setups and encryptions counts. Every discovered error was fixed and the last 100 configurations ran error-free.

The functionalities of the plugin not covered by this test, e.g., file saving and oscilloscope communication, were tested manually.

Switch	**Default value**	**Description**
-m	none	Mode of operation: random - generate DEFINITIONS.vhd randomly; fixed - generate DEFINITIONS.vhd using config.txt file.
-t	none	Run the synthesize-translate-map-place&route-generate programming file toolchain for both FPGAs.
-p	none	Download the programming file to both FPGAs. Impact is run in the batch mode with *impact_download_fpga.txt* as the file argument. Ports have to be set in the file first.
-l	none	Run SICAK *meas' sakurag* plugin in test mode with all of the possible parameter values for every input.
-a	none	Run the whole test sequence — same behaviour as with -t -p -l.
-r	1	Number of test sequences to be run. Only works with -a or -t -p -l, otherwise it is one.
-s	random	Seed value. When an error is discovered, the seed is saved to test_errlog.txt, and the same run can be repeated using this option.
-n	100	Number of encryptions to be run (SICAK *meas'* -n parameter).

Table 5.1: test.py script options.

The size of the Control FPGA design depends on the input/output count, width and number of shares. Example usage of Control FPGA's slice registers and Look-up tables (LUTs) is in the table 5.2. The Control FPGA of Sakura-G has 11440 slice registers and 5720 LUTs.

Configuration	**Slice registers**	**LUTs**
128/128/7/7	5247	5370
128/128/8/8	-	-
128,128,128/128/3,2,1/1	3491	3552
128,128,128/128/4,2,1/1	4132	4831
128,128,128/128/5,2,1/1	4645	5653
256/256/4/4	6412	5114
256/256/5/5	-	-
256,256/256/2,2/1	4623	2732
256,256/256/3,3/1	6160	4345
256,256/256/4,3/1	-	-
256,256,256/256/3,1,1/1	5655	4053
256,256,256/256/3,2,1/1	6167	4725
256,256,256/256/3,2,2/1	-	-

Table 5.2: Examples of slice registers and Look-up tables usage of the synthesized Control FPGA design. The left Configuration is input width 0, .., input width I/output width 0, .., output width J/input shares count 0, .., input shares count I/output shares count 0, .., output shares count J. The dash symbol means that the configuration could not be mapped into the FPGA. The Control FPGA of Sakura-G has 11440 slice registers and 5720 LUTs.

The duration of a measurement is more than a hundred times reduced. For example, measurement of 100000 traces using Fixed vs. random test took 7 seconds instead of 1608 seconds when a cipher wrapper which has to receive/send every input/output from/to the computer separately was used. Running the same setup without oscilloscope requires 2 seconds, therefore the bottleneck is the measurement files saving and/or the communication between the oscilloscope and the computer.

Chapter **6**

Conclusion

In this thesis, a toolkit for fast data acquirement during the process of power analysis was created. Sakura-G board was used and a configuration for both of its FPGAs was designed in VHDL language. The Main FPGA design receives the data from the Control FPGA and manages the cipher instantiated inside of it. The Control FPGA design minimizes communication with the computer by generating the input data pseudo-randomly. It sends and receives the data from the Main FPGA. It supports various run modes — Fixed mode, where the input data remain constant; Random mode, where the data are randomized before every encryption; Fixed vs. random mode where the input data before every encryption are either constant or random in a random order; and FTDI mode, where the data are received from the computer before every encryption. It can also optionally remask the input data. Output data can be sent to the computer.

A plugin for the Software Toolkit for Side Channel Attacks (SICAK) was created in C++ language using the Qt5 framework. The plugin handles the communication with the Control FPGA. It reproduces every input that was created inside of the FPGA device independently; it can receive the output data or the output data can be reproduced without communication when a software implementation of the cipher instantiated in Main FPGA is provided.

The toolkit supports ciphers with an arbitrary number of input/output counts and an arbitrary number of shares per input/output. The VHDL FPGA design is highly generic and a constant package is generated automatically using a simple configuration file with user-filled values. The maximum total number of input/output shares that can fit into the Control FPGA of Sakura-G is about 7/7 for 128 bit wide inputs/outputs and 4/4 for a 256 bit wide inputs/outputs.

Duration of measurement improved more than 100×. For example, a measurement of 100000 power traces took 7 seconds. With a setup that has to communicate with the computer during the whole operation, the same measurement took 1608 seconds.

The FPGA design was thoroughly verified using unsynthetizable models and constrained-random inputs. A random configuration generator was created to test the design with as many different configurations as possible.

The whole system was thoroughly tested. A script automatizing the creation of new configurations for the FPGA and its downloading to the FPGA was created. The system was tested with many different configurations using various run modes.

Bibliography

[1] Brier, E.; Clavier, C.; Olivier, F. "*Correlation Power Analysis with a Leakage Model*". In: Volume 3156 of the book series Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2004. DOI: 10.1007/978-3-540-28632-5_2.

[2] Computer Systems Laboratory, Washington University. "*The Synchronizer "Glitch" Problem*". In: Macromodular Computer Design, Part 1, Volume 4 (1974).

[3] Future Technology Devices International Ltd. *FT2232H Dual High Speed USB to Multipurpose UART/FIFO IC datasheet [online]*. 2019. URL: https://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT2232H.pdf.

[4] George, M.; Hafke, P. *Linear Feedback Shift Registers in Virtex Devices [online]*. 2007. URL: https://www.xilinx.com/support/documentation/application_notes/xapp210.pdf.

[5] Kocher, P.; Jaffe, J.; Jun, J. "Introduction to differential power analysis - Crypto 99 Proceedings, Lecture Notes in Computer Science Vol. 1666". In: (1999).

[6] Mangard, S.; Oswald, E.; Popp, T. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer-Verlag, 2007. ISBN: 978-0-387-30857-9.

[7] Mentor, a Siemens Business. *Modelsim PE Student Edition*. Version 10.4a. URL: https://www.mentor.com/company/higher_ed/modelsim-student-edition.

[8] Morita Tech Co. *Sakura-G - Side-channel AttacK User Reference Architecture [online]*. 2013. URL: http://satoh.cs.uec.ac.jp/SAKURA/hardware/SAKURA-G_Spec_Ver1.0_English.pdf.

[9] Pico Technology Ltd. *Picoscope 6000 Series Datasheet [online].* URL: https : / / www . picotech . com / download / datasheets / PicoScope6000CDSeriesDataSheet.pdf.

[10] Schneider, T.; Moradi, A. "*Leakage Assessment Methodology - a clear roadmap for side channel evaluations*". In: Journal of Cryptographic Engineering (2016). DOI: https://doi.org/10.1007/s13389-016-0120-y.

[11] Socha, P. *Software toolkit for side-channel attacks. Master's thesis.* Technical University in Prague, Faculty of Information Technology, 2019.

[12] The Qt Company. *Qt5 Framework*. Version 5.9.5. URL: https://www.qt.io/.

[13] Tristan Gingold. *GHDL, the open source VHDL simulator*. Version v0.37. URL: https://github.com/ghdl/ghdl.

[14] Welch, B. L. "*The generalization of "Student's" problem when several different population variances are involved*". In: (1947). DOI: https://doi.org/10.1093/biomet/34.1-2.28.

[15] Xilinx, Inc. *ISE WebPACK design software*. Version 14.7. URL: https://www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.html.

Appendix A

List of Acronyms

AES Advanced Encryption Standard

CPA Correlation Power Analysis

DPA Differential Power Analysis

FPGA Field Programmable Gate Array

FSM Finite State Machine

FTDI Future Technology Devices International Ltd.

LFSR Linear-Feedback Shift Register

LUT Look-Up Table of FPGA

PRNG Pseudo-Random Number Generator

UART Universal Asynchronous Receiver/Transmitter

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

Appendix B

Attached DVD description

```
readme.txt ................................ DVD structure description
├── ISE
│   ├── CONTROL ................................ ISE project for Control
│   └── MAIN ....................................... ISE project for Main
├── sicak-plugin .................................... The SICAK plugin
├── VHDL
│   ├── CONTROL........................................Control entities
│   ├── MAIN ............................................ Main entities
│   ├── TB ................................................ TB entities
│   ├── common ............................... common design entities
│   └── tmpl .................... template for DEFINITIONS generation
├── misc
│   ├── dia .................................. diagram files for Dia SW
│   ├── graphviz ................ source codes for FSM diagrams generation
│   └── wavefrom ....................... source codes for Wavedrom SW
└── run
    ├── config.txt ...... the configuration file for DEFINITIONS generation
    ├── generate_definitions.py .. generate DEFINITIONS from config.txt
    ├── verify.py.................................... verification script
    ├── test.py.......................................... testing script
    ├── run_cmd.do.......................... list file for MODELSIM CLI
    ├── run.do.............................. list file for MODELSIM GUI
    ├── source.lst................................... list file for GHDL
    └── impact_download_fpga.txt...................... Impact batch file
```