D4.1: Cryptographic Primitives Prototype Implementation

Project number:	238811
Project acronym:	**UNIQUE**
Project title:	Foundations for Forgery-Resistant Security Hardware
Start date of the project:	01.09.2009
Duration:	30 months

Deliverable type:	Document
Deliverable reference number:	238811/ D4.1 / v1.0
Deliverable title:	Cryptographic Primitives Prototype Implementation
WP contributing to the deliverable:	WP4
Due date:	2011-08-31 (M24)
Actual submission date:	2011-08-31 (M24)

Responsible organisation:	KULEUVEN
Authors:	Vincent van der Leest, Erik van der Sluis, Peter Simons (all IID), Roel Maes, Anthony van Herrewege (all KULEUVEN), Timm Korte (SIRRIX)
Abstract:	This deliverable contains a detailed description of the implementation of cryptographic primitives required for the prototype use cases within the UNIQUE project. The description contains hardware set-up and building blocks.
Keywords:	IP modules, demonstrators

Dissemination level:	Public
Revision:	V1.0

Instrument:	STREP
Thematic Priority:	ICT

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Scope of document

This deliverable contains a detailed description of the implementation of cryptographic primitives required for the prototype use cases within the UNIQUE project. The description contains hardware set-up and building blocks. This way the deliverable provides the reader with a complete overview of the implementation and interfaces of the separate building blocks and their use in the implementation of the UNIQUE demonstrators.

1.2 List of abbreviations

ASIC	Application Specific Integrated Circuit
DFF	Data Flip-Flop
FPGA	Field Programmable Gate Array
I/O	Input / Output
LR-PUF	Logically Reconfigurable PUF
MPW	Multi Project Wafer
NVM	Non-Volatile Memory
PUF	Physically Unclonable Function
RFID	Radio Frequency IDentification
RO	Ring Oscillator
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
TSMC	Taiwan Semiconductor Manufacturing Company

1.3 Document overview

In Chapter 2, the requirements for the software part of the development environment are listed while Chapter 3 defines the implementation of the cryptographic primitives developed for the UNIQUE prototype.

2 Development environment requirements

The following is a list of software required to synthesise and simulate the FPGA cryptographic blocks necessary for the UNIQUE use cases. When possible, free software has been preferred over commercial packages. All listed software can be made to work on both Windows & Linux, and thus by extension should probably work on MacOS.

Software name	Description	Free
Xilinx ISE Webpack	Synthesis of HDL designs for Xilinx FPGAs	X
Icarus Verilog	Terminal-based Verilog simulator	X
GTKWave	Waveform viewer for simulation results	X
ModelSim SE	GUI-based Verilog & VHDL simulator	

Table 1: List of required software packages

3 Cryptographic primitives

In this section, we describe the cryptographic primitives created for the use cases. In the first subsection, "Block usage", a high level architecture for both use cases is provided and a concise description of each block's behaviour is given, detailing how to drive the inputs and how results will appear on the output ports.

The second subsection, "Block specifications", provides a more formal specification for each block implementation, listing the HDL language used, the input and output ports, configurable parameters and the internal architecture of each block.

Furthermore, an estimate is given for how much slices the block occupies in a Virtex-5 XC5VLX50 and the maximum speed at which it can run. Notice that these two numbers heavily depend on the synthesis goals and the type of FPGA used and thus should be taken only as a very rough estimate. The maximum speed is for a design with the given number of slices.

3.1 Block usage

3.1.1 Use case 1

A global overview of the implementation architecture of use case 1 is given in the image below. Unless noted otherwise, all designs work with rising clock edge sensitive flip-flops.

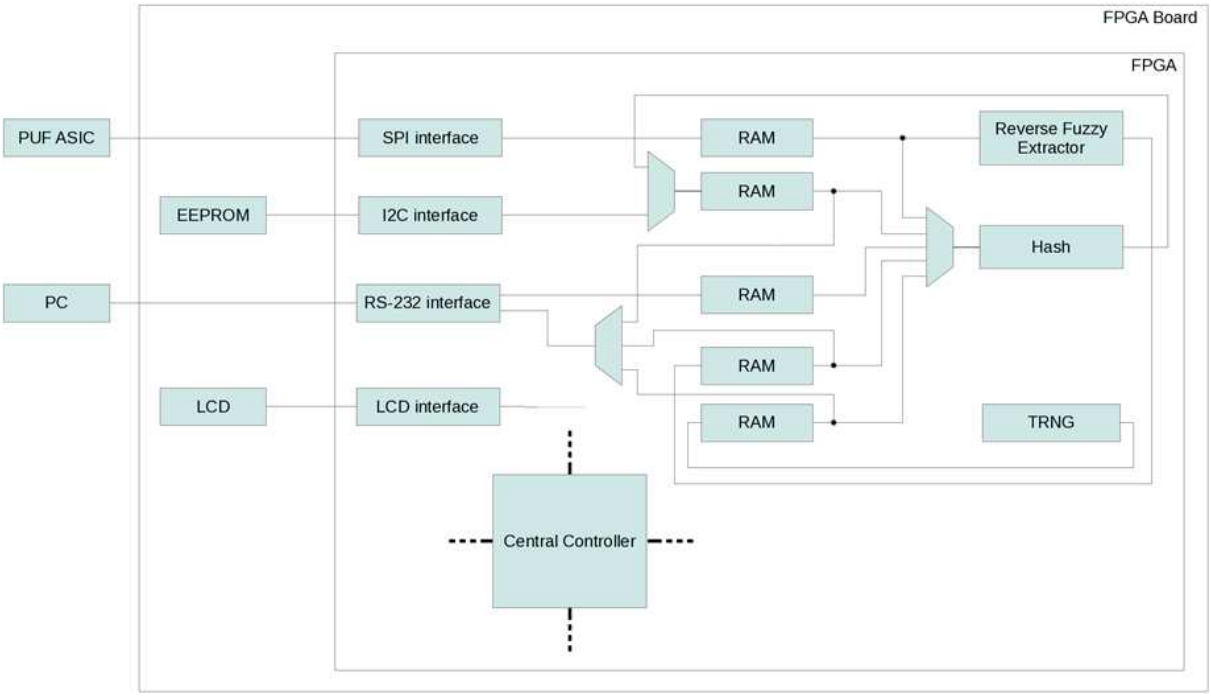


Figure 1: Use Case 1 - Recyclable Tokens

3.1.1.1 True Random Number Generator (TRNG)

As long as the enable input of the block is driven high, a new randomly generated bit appears on the TRNG port every clock cycle. No initialisation is necessary. Note that this block is always running, the only thing the enable signal controls is the sampling of the output register.

3.1.1.2 Reverse fuzzy extractor

This block implements the generation phase of a syndrome-construction based fuzzy extractor. It calculates helper data on a PUF response as a multiplication of the response with a parity-check matrix of a linear block code. This is done efficiently using an LFSR-based implementation.

To add a new bit as input, put the value on the data_in port and make enable high. The data on the data_out port will be updated accordingly. Depending on the selected data_out port width and the size of the syndrome polynomial, one will need to select which part of the syndrome is put on the data_out port by changing the value of address_sel, which controls the MUX that routes the syndrome data chunks.

3.1.1.3 SPONGENT core (Hash)

This is a straightforward, pipelined implementation of the SPONGENT hash function. At the time of this writing, this function has not officially been published yet. The implementation was compared with a preliminary implementation by the designer of the SPONGENT algorithm.

To use the hash function, first reset it, by toggling reset high. Next, put message data on the data_in port and make msg_data_available high. Then make start_continue high. In response, busy will go high and the SPONGENT round function will be executed an appropriate number of times. Afterwards, busy will go low and new message data will be processed if start_continue and msg_data_available are high.

Once all message data hash been processed, the resulting hash can be gotten by making start_continue high, but keeping msg_data_available low. The block will then execute a squeeze phase and once busy is low, a number of bits of the hash appear on the data_out port. Repeat this as many times as necessary to get the full hash output.

3.1.2 Use case 2

The location of the cryptographic primitives for use case 2 is shown in the following picture. The clock, reset and flush signals are global signals and not included in this picture.

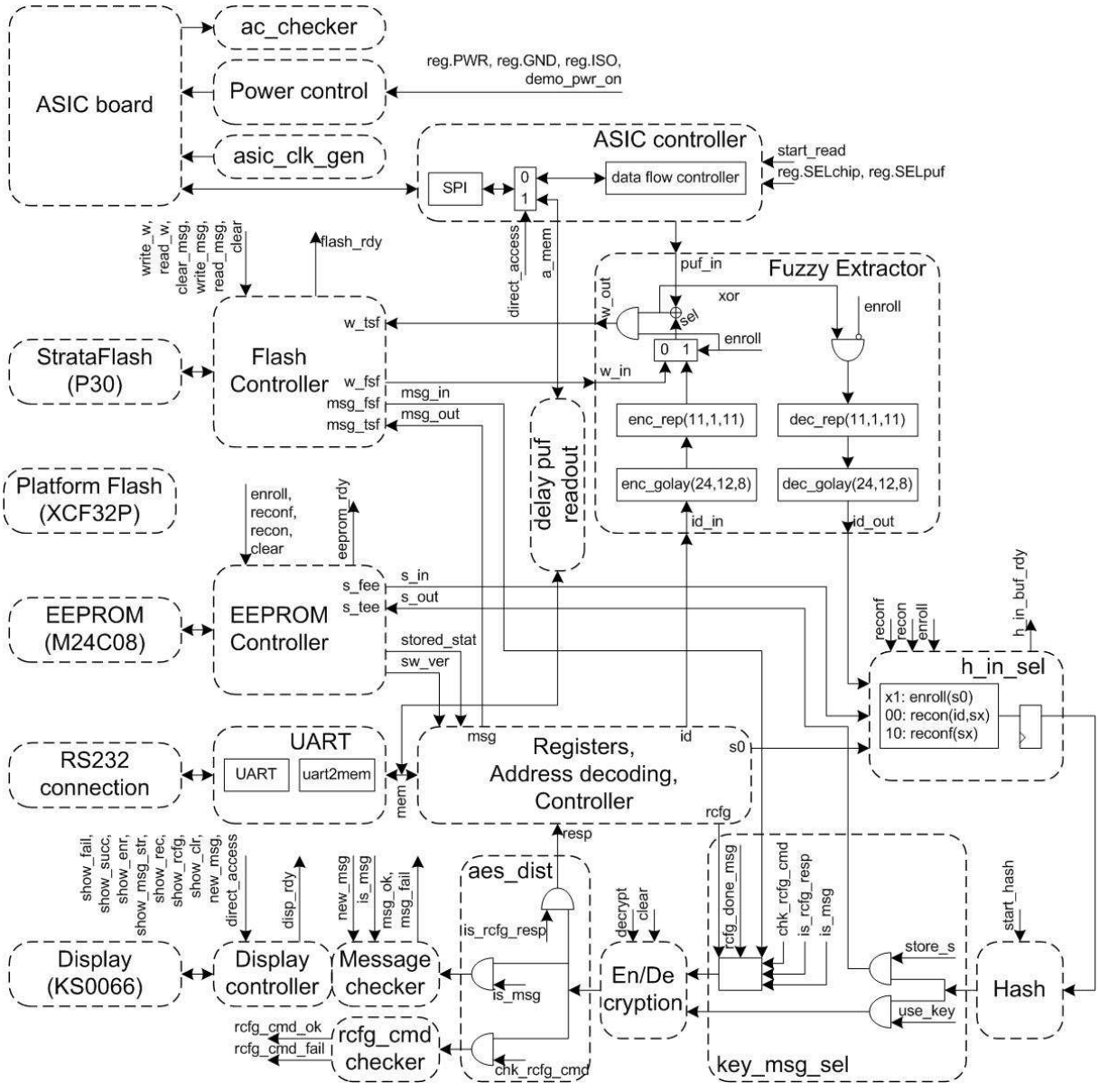


Figure 2: Use Case 2 - HW/SW-binding

3.1.2.1 Data input and output interfaces

All data interfaces have three signals:

- *_d: This carries the actual data. For the different connections between the blocks the data bus widths can be different
- *_or: Via this signal the data source block indicates that the data is valid and can be copied
- *_ir: Via this signal the data receiver block indicates that it has copied the data and new data can be put on the data bus.

Every time the or signal and the ir signal are 1 at the same time the data is copied on the next positive clock edge.

3.1.2.2 AES core

This module implements AES with 128 bit keys and 128 bit data blocks. The key and data interfaces are 128 bits.

The module first asks for a key. When that is provided it calculates the key schedule based on the value of decrypt. Then it asks for the data and encrypts or decrypts each data block.

When the mode must be changed between encrypt and decrypt or a new key must be used the clear pin must be set to 1 for at least one clock cycle. After that the module asks again for a key, calculates the key schedule, etc.

3.1.2.3 Hash module

This module collects the required values based on the enroll, recon and reconf signals (only one can be active at a time), stores this in a buffer and provides this to the hash module. The generated values are shown in the following table.

- id field is 171 bits long
- s0 field is 256 bits long
- sx field is 256 bits long
- <length> field is 64 bits long
- The 0..0 field fills the remaining space before the <length> field with 0 so that the entire length is 512 bits

Signal	Output
enroll	0xFFFFFFFF & s0 & 1 & 0..0 & <length = 288>
recon	id & sx & 1 & 0..0 & <length = 427>
reconf	sx & 1 & 0..0 & <length = 256>

Table 2: Hash module values

The hash module implements the SHA-256 specification for one block. Padding and addition of the length is done in the h_in_sel module.

3.1.2.4 Fuzzy extractor

During enrollment (enroll = 1) the fuzzy extractor receives data from the id_in interface, gathers blocks of 12 bits and Golay-encodes this into 24 bits. These are repetition encoded with a repetition of 11. The output of this is XOR'ed with the bits received on the puf_in interface, and that is passed to the w_out interface.

During reconstruction (enroll = 0) the helper data is received from the w_in interface. These bits are XOR'ed with the bits received on the puf_in interface and passed to the repetition decoder. This corrects the input data and outputs 1 bit per 11 input bits. The Golay decoder gathers 24 bits and corrects that into 12 bit words that are provided on the id_out interface.

The fuzzy extractor is built with 4 blocks (enc_golay, enc_rep, dec_rep, dec_golay) and some glue logic to select the data paths for enrollment or reconstruction.

3.2 Block specifications

For each block, ports are specified as follows:

port_name [port_width direction]: description

Where direction is one of:

- i: input
- o: output
- io: input & output

3.2.1 Use case 1

3.2.1.1 True Random Number Generator (TRNG)

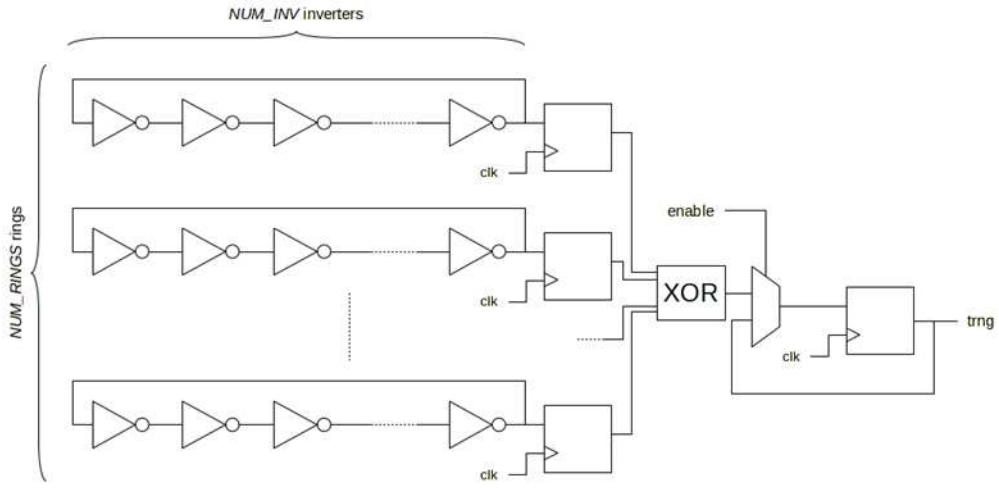
Language	Verilog 2001
Description	A true random number generator based on synchronously sampled oscillator rings. Taken from [1] 385-390.
Ports	• clk [1 i]: clock input • enable [1 i]: active high enable input • trng [1 o]: true random output
Parameters	• NUM_RINGS: number of oscillator rings • NUM_INV: number of inverters per oscillator ring
Schematic	

Table 3: True Random Number Generator

3.2.1.2 Reverse fuzzy extractor

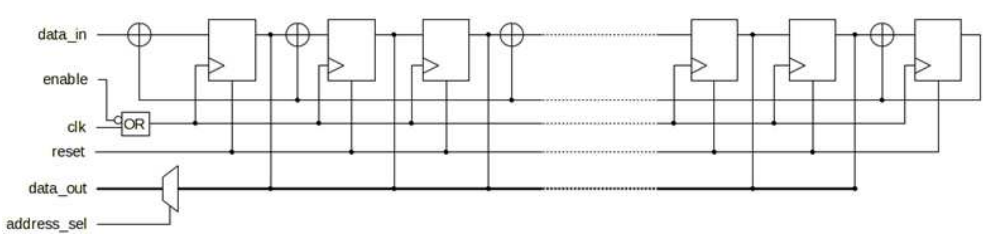
Language	Verilog 2001
Description	Reverse fuzzy extractor for PUF data
Ports	• ckl [1 i]: clock signal • reset [1 i]: active high reset signal • enable [1 i]: enable signal • data_in [1 i]: input bit • data_out [x i]: data output (for width, see parameters) • address_sel [log2(x) i]: address selection for data_out MUX
Parameters	• SYNDROME_POLY: polynomial to use for syndrome generation • SYNDROME_DATA_OUT_WIDHT: length of SYNDROME_POLY – 1 • DATA_OUT_WIDTH: width of data_out port
Schematic	

Table 4: Reverse fuzzy extractor

3.2.1.3 SPONGENT core (Hash)

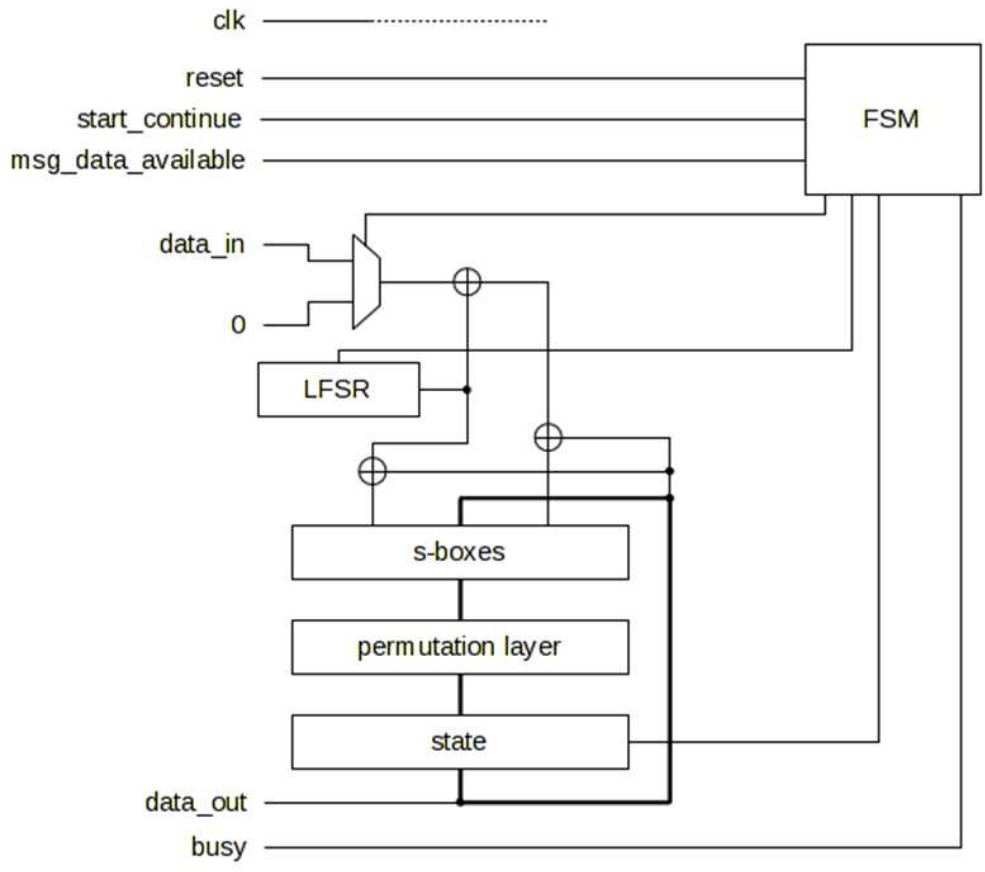
Language	Verilog 2001
Description	SPONGENT hash core
Ports	• ckl [1 i]: clock signal • reset [1 i]: active high reset signal • busy [1 o]: busy signal • start_continue [1 i]: start/continue hashing signal • msg_data_available [1 i]: absorb or squeeze signal • data_in [x i]: message data input (for size, see parameters) • data_out [x o]: hash data output (for size, see parameters)
Parameters	• STATE_SIZE: size of internal hash state • RATE: data_in & data_out port width • LFSR_POLY: polynomial for internal LFSR • LFSR_INIT: initialisation value for LFSR Note: only tested with official values for SPONGENT-128.
Schematic	

Table 5: Spongent core (Hash)

3.2.2 Use case 2

3.2.2.1 AES Core

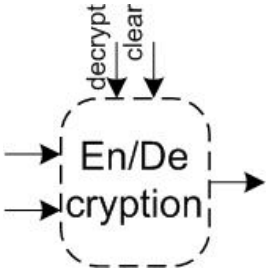
Language	VHDL
Description	AES core for 128 bit keys, ECB mode
Ports	• arst_n [1 i]: Reset, active low • clk [1 i]: clock • clear [1 i]: clear the AES core and the key • decrypt [1 i]: 1: Decrypt function; 0: encrypt function • key_d [128 i]: Key input • key_or [1 i]: Key input valid • key_ir [1 o]: Key input accept • in_d [128 i]: Data input • in_or [1 i]: Data input valid • in_ir [1 o]: Data input accept • out_d [128 o]: Data output • out_or [1 o]: Data output valid • out_ir [1 i]: Data output accept
Parameters	none
Schematic	

Table 6: AES Core

3.2.2.2 Hash module

Language	VHDL
Description	Hash module specific for the use case 2 hashes; implements SHA256 for one 512 bit block It is built with two modules: • h_in_sel (creating the 512 bit hash input) • hash (implementing SHA256 for one block)
Ports	• arst_n [1 i]: Reset, active low • clk [1 i]: Clock • enroll [1 i]: Create hash of S0 • recon [1 i]: Create hash of ID and Sx • reconf [1 i]: Create hash of Sx • flush [1 i]: Clear the has state • start_hash [1 i]: Start hashing • hash_rdy [1 o]: Hash is finished (not used) • h_in_buf_rdy [1 o]: Hash input data is buffered (not used) • id_out_d [12 i]: ID input data • id_out_or [1 i]: ID input valid • id_out_ir [1 o]: ID input data accept • s0_d [32 i]: S0 input data • s0_or [1 i]: S0 input valid • s0_ir [1 o]: S0 input accept • s_in_d [32 i]: Sx input data • s_in_or [1 i]: Sx input valid • s_in_ir [1 o]: Sx input accept • hout_d [32 o]: Hash output data • hout_or [1 o]: Hash output valid • hout_ir [1 i]: Hash output accept Middle interface Interface as seen on the hash module • hin_d [32 i]: Data • hin_or [1 i]: Data valid • hin_ir [1 o]: Data accept
Parameters	None

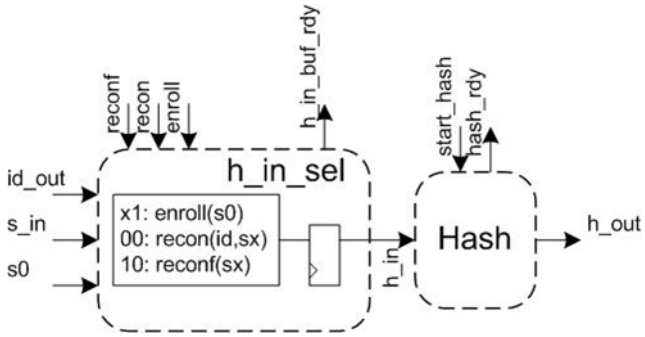
Schematic	

Table 7: Hash module

3.2.2.3 Fuzzy extractor

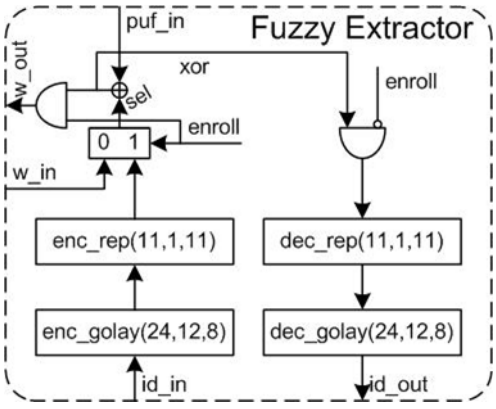
Language	VHDL
Description	Fuzzy extractor, encoder and decoder, using golay(24,12,8) and repetition(11, 1, 11)
Ports	• arst_n [1 i]: Reset, active low • clk [1 i]: Clock • enroll [1 i]: 1: Enrollment flow; 0: Reconstruction flow • flush [1 i]: Clear all states • puf_in_d [1 i]: PUF input data • puf_in_or [1 i]: PUF input valid • puf_in_ir [1 o]: PUF input accept • id_in_d [12 i]: ID input data • id_in_or [1 i]: ID input valid • id_in_ir [1 o]: ID input accept • w_out_d [1 o]: Helper data output • w_out_or [1 o]: Helper data output valid • w_out_ir [1 i]: Helper data output accept • w_in_d [1 i]: Helper data input • w_in_or [1 i]: Helper data input valid • w_in_ir [1 o]: Helper data input accept • id_out_d [12 o]: ID output data • id_out_or [1 o]: ID output valid • id_out_ir [1 i]: ID output accept
Parameters	None
Schematic	

Table 8: Fuzzy extractor

4 References

[1] K. Wold and C. How Tan. *"Analysis and Enhancement of Random Number Generator in FPGA Based on Oscillator Rings."* In Proceedings of RECONFIG'2008. pp.385~390.