CERG SHA3 Core Documentation

Summary

This document describes the functionality of GMU's implementation of the SHA-3 family of hash functions. The implementation is written entirely in VHDL and is available on GitHub at the link listed below.

The Implementation supports all modes of the SHAKE XOF (SHAKE128 AND SHAKE256) and the two most common SHA-3 hash modes (SHA3-256 and SHA3-512). This is a high-performance design that performs one round the internal Keccak round function per cycle. Since 24 rounds are performed per permutation, one permutation requires 24 cycles. Data and commands are both loaded into the SHA3 module through a single 64-bit stream interface. Data is received from the core through another 64-bit stream interface. The core cannot be parameterized; only this input format is supported.

GitHub Link:
https://github.com/GMUCERG/SHAKE

SHA3 Signals

The table below briefly describes all interface signals of the SHA3 module. The mode, input length, and output length of the current operation are set by the first word of input sent through the `din` port. The format of the command word and data input is specified in the following section.

The input and output data signals use a ready/valid handshake. Every transaction begins with a single command words passed through the `din` port, followed by the input data also passed through the `din` port. After all input data has been loaded into the module, the hash result is unloaded through the `dout` port.

The user sets `src_ready` low when `din` holds valid data. The core asserts `src_read` high when it has accepted the input data. Similarly, the user sets `dst_ready` low to signal that output is ready to be received, and the core asserts `dst_write` high when `dout` is valid.

Signal	**Direction**	**Polarity Description**	
rst	input	high	Initializes the design
clk	input	N/A	Input clock signal
src_ready	input	Low	Assert low when din data is valid
src_read	output	High	Asserted high to signal that din data has been accepted

Signal	Direction	Polarity Description	
dst_ready	input	Low	Assert low to accept dout data
dst_write	output	High	Asserted high when dout data is valid
din[63:0]	input	N/A	Data and command input
dout[63:0]	output	N/A	Data output

SHA3 Command Format

The data below briefly describes the SHA3 module command format.

When the desired output length is unknown at the time of issues the command, set the dout size to the maximum value and then reset the core when the operation is completed.

din[63:60]	din[59:32]	din[31:0]
Operation Mode: `0xC`->SHAKE128 `0x8`->SHA3-256 `0xA`->SHA3-512 `0xE`->SHAKE256	Output data size in bits	Input data size in bits

SHA3 Data Format

Data is loaded in little-endian form, so if only 6 bytes $b_0, b_1, \ldots, b_5$ are in the final word of input, din would be formatted as follows:

din[63:56]	din[55:48]	din[47:40]	din[39:32]
b_5	b_4	b_3	b_2

din[31:24]	din[23:16]	din[15:8]	din[7:0]
b_1	b_0	0x00	0x00

Latency Formula

The performance of the SHA3 module depends on the rate of the SHA3 mode. The rate of the supported modes of SHA-3 are listed below.

Algorithm	Rate `r` (Bytes)
SHA3-256	136
SHA3-512	72
SHAKE128	136
SHAKE256	168

Internally, there are three buffers: input, hash, and output. The first block of input to the next operation can be loaded while the current operation completes. Similarly, while an output block is being unloaded, the permutation to calculate the next block of output is performed.

Thus, when processing multiple blocks of input and output, the inner loading and unloading cycles are masked by the permutation latency. The total latency is then the sum of the latency of the first block loaded, the number of permutations required to ingest the input data, the number of permutations required to squeeze the output data, and the number of cycles required to unload the last bytes of output data.

The input data must be shifted serially into place, so loading always requires `r/io_width` cycles. The permutation requires `perm_cc=26` cycles for the 24 rounds of the permutations plus one cycle to finish and one cycle to unload the state into the output buffer.

The latency for `din_bytes` of input and `dout_bytes` of output can be calculated as follows:

Latency For a Single Hash:
`latency = r/io_width + perm_cc*ceil(din_bytes/r) + perm_cc*floor(dout_bytes/r) + (dout_bytes%r)/io_width`

- `io_width = 8`
- `perm_cc = 26`

Running the SHA3 Example Testbench

- Clone or download the VHDL implementation from https://github.com/GMUCERG/SHAKE
- Create a Vivado project and add all source files under the folder `src_all` as design sources
- Add all sources under `tb` as simulation sources.
- Update lines `29` and `30` in the test files `sha_tb_all.vhd` to point to the test files `kat/kat_all/ALL_ZERO_ALL_VERSION_IN.txt` and `kat/kat_all/ALL_ZERO_ALL_VERSION_OUT.txt`.

You can now run the simulation. This testbed will run a number of tests for various SHA3 modes.