SHA-256 Hash Generator in Verilog HDL

Bartosz Rulka[1,2] , Paweł Pieńczuk[1,2], Witold Pleskacz[2]
[1] Łukasiewicz Research Network – Institute of Microelectronics and Photonics, Warsaw, Poland
[2] Institute of Microelectronics and Optoelectronics, Warsaw University of Technology, Warsaw, Poland
e-mail: bartosz.rulka@imif.lukasiewicz.gov.pl

***Abstract*—An implementation of SHA-256 hash generator is presented. A block has been described in Verilog HDL. A generator code is written with basic logical and arithmetic operations to create a easily-synthesizable block. A design process a 512-bit block in 67 cycles. The generator is tested in simulations with the test vectors and reference digests published by NIST. The simulation testbench has been designed in SystemVerilog. The design passed all test cases used.**

***Keywords*—hash function, SHA-256, HDL, Verilog.**

Extended Abstract

A SHA-256 is one of the SHA-2 hash function family participant published by U.S. National Institute of Standards and Technology (NIST). Thanks to the hardware implementation of this cryptographic function, the compatibility of uploading software, transactions, etc. can be checked.The SHA-256 generator block is designed in Verilog HDL using the SHA-256 hash function. Generator after serving on the data input to be hashed and the length of the message binary record returns to output *hash* with a length of 32 bytes (256 bits).

The generator is created with 21 interconnected submodules that execute appropriate logical and arithmetic operations (like XOR, OR, AND, bit-shift, +, etc.) that control and operate the course of subsequent rounds across the algorithm. The built-in controller controls the block submodules in compliance with the *FIPS 180-4* standard. The entire block is resetable by asynchoronous *RESET* input.

The generator starts processing at the rising edge of *CLK* when the *FLAG_IN* flag is raised. It confirms that 512-bit (one chunk) of input data *MESSAGE* and 64-bit input data length were sent. Immediately, the number of 512-bit data blocks is computed by division of *MESSAGE_LENGTH* by $2^9 = 512$ (realized by the 9-bit shift). This *OUT_CHUNK* output value is needed by the controller to control how many of the chunks the generator needs to compute.

Preprocessing step is split into two steps, Parsing and Padding. We decided to implement the parsing step in software, but the padding inside hardware implementation. The preparation of data and padding is done within three clock cycles, but the padding itself is done in one clock cycle. However, these steps are rather complex since these require e.g. bit rotation by variable bit number.

Our preliminary synthesis and static timing analysis (STA) tests with several CMOS standard libraries shown that the critical path is mostly hidden in padding operation. This issue needs further investigations, but we see two potential approaches to deal with it:

- pipelining the operation in preprocessing;
- move the preprocessing operations to the software domain.

Main SHA256 loop block strictly follows the *FIPS 180-4* standard. Since the SHA-256 algorithm is designed with simple arithmetic and logical operations (AND, OR, ROTR, etc.), the implementation in any HDL is rather straightforward.

After the preprocessing, the following operations are performed

- $H_{0:7}^{(0)}$ values initialization;
- $W_t^{(i)}$ message schedule preparation;
- main compression loop;
- intermediate hash $H_{0:7}^{(i)}$ update;

After processing a 512-bit block, the intermediate hash values are used in the next round (with latter 512-bit data block). When the entire message is processed, the final 8 32-bit hash values are concatenated to the final 256-bit output hash value. Simultaneously, the output flag *FLAG_OUT* is raised, indicating the end of the calculations. Now the outer system can collect the output hash. The generator computes one 512-bit chunk of data in 67 clock cycles.

The generator tests start with reading all test vectors from the text files provided by *NIST*.

Then, these vectors are transmitted to the *MESSAGE* input, and the length of their binary notation to the *MESSAGE_LENGTH* input. After the calculations, the testbench save the generator's output hash value. After that, the calculated hash is compared to the reference hash *MD (Message Digest)*. If the hashes are the same, the *SUCCESS* message is written in the log file. Otherwise, the *ERROR* message is written in the log file. The project can be extended by adding the standard interface (such as Wishbone or AXI4-Lite) to improve the adaptability to microprocessor architecture and verify a design on an FPGA platform to perform experimental tests.