

Hardware Interface of a Secure Hash Algorithm (SHA)

Functional Specification

v. 1.3 October 17, 2009

by CERG at George Mason University

Below we provide our proposal for a standard hardware interface of a secure hash algorithm, SHA. Our interface supports hash function SHA-1 and the family of hash functions SHA-2 described in [1]. The same interface will most likely apply to all (or at least majority) of hash algorithms competing in the contest for the new SHA-3 standard.

1. Input/output ports

In Fig. 1, we illustrate the names, directions, and widths of input and output ports of the proposed SHA core.

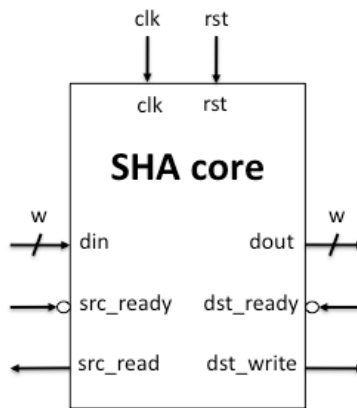


Fig. 1: Input/output interface of a SHA core

Table 1 describes functions of all SHA ports. The constant parameter w represents the natural data word width for a given function. In particular, $w=32$ for SHA-1, SHA-224, and SHA-256; $w=64$ for SHA-384 and SHA-512.

Table 1: Functionality of Input/Output Ports

Global Control Signals		
clk	in	Global clock.
rst	in	Global reset, active HIGH. Minimum duration equal to one clock cycle. After the positive reset pulse, the SHA core becomes ready to hash a new message. In case the reset appears during hashing of a message, the hashing is abandoned, no output is generated, and the core becomes ready to accept a new message.

Input Data Interface		
din[w-1:0]	in	A w-bit word of input data.
src_ready	in	A control signal indicating that the source of input is ready to be read from. Active LOW.
src_read	out	A control signal used to read data from the source of data. Data from the din input is assumed to be stored in the SHA core at the next rising edge of the clock. Active HIGH.
Output Data Interface		
dout[w-1:0]	out	A w-bit word of output data (i.e., a word of a hash value).
dst_ready	in	A control signal indicating that the destination of output is ready to be written to. Active LOW.
dst_write	out	A control signal used to write data to the destination of data. Data from the dout output is assumed to be accepted by the destination circuit at the next rising edge of the clock. Active HIGH.

2. Typical scenario

In a typical scenario, the SHA core is assumed to be surrounded by two standard FIFO modules: Input FIFO and Output FIFO, as shown in Fig. 2.

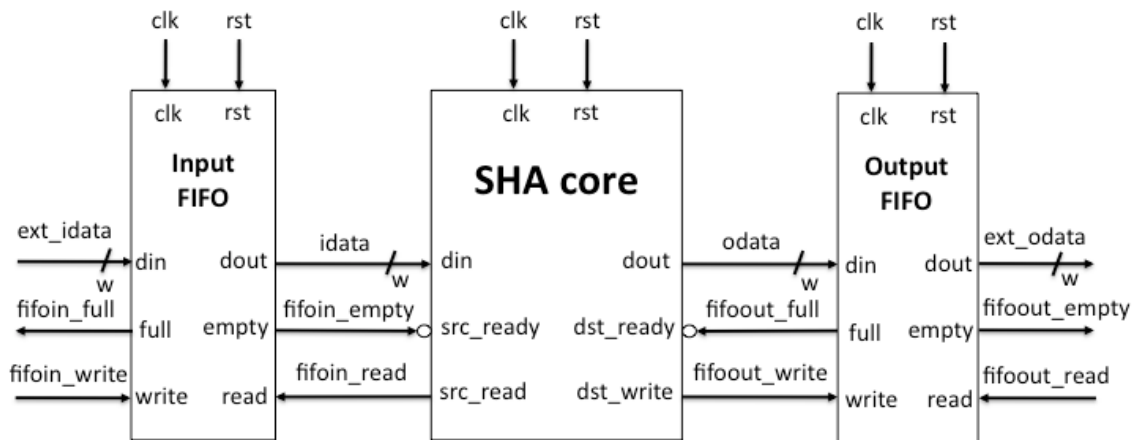


Fig. 2: A typical configuration of a SHA core connected to two surrounding FIFOs. The Input FIFO serves as a source of input data, and the Output FIFO as a destination for output data.

Each FIFO module generates signals empty and full, which indicate that the FIFO is empty and/or full, respectively. Each FIFO accepts control signals write and read, indicating that the FIFO is being written to and read from, respectively.

After a global reset (active value of the signal `rst`) both FIFOs are assumed to be empty. As soon as data is written to Input FIFO, the control output of this FIFO, `fifoin_empty`, becomes low, which is an active value of the SHA port `src_ready`. This is a signal for the SHA core to activate `src_read` and thus `fifoin_read`. The `fifoin_read` signal is active in every clock cycle in which SHA core is ready to accept a new word of data and the `fifoin_empty` signal is inactive.

After processing the entire message, SHA core writes a hash value to Output FIFO one word per clock cycle. In order to do that, SHA core activates an output port `dst_write` connected to `fifoout_write`. This output remains active until either the hash value is completely written to Output FIFO, or this FIFO is temporarily full. In the latter case, the transfer of output data to the Output FIFO will resume as soon as `fifoout_full` is low.

The aforementioned assumptions about the use of FIFOs as surrounding modules are very natural and easy to meet.

If a SHA core implemented on an FPGA communicates with an outside world using PCI, PCI-X, or PCIe interface, the implementations of these interfaces most likely already include Input and Output FIFOs that can be directly connected to the SHA core.

If a SHA core communicates with another core implemented on the same FPGA, then FIFOs are often used on the boundary between the two cores in order to accommodate for any differences between the rate of generating data by one core and the rate of accepting data by another core.

In the described above configuration, SHA core is an active module, while a surrounding logic (FIFOs) is passive. Passive logic is much easier to implement, and in our case is composed of standard logic components, FIFOs, available in any major library of IP cores. Performance and area overhead associated with the use of FIFOs can be quite easily controlled, depending on available resources. In particular, the FIFOs can have arbitrary depth, including the depth of one, in which case, they may be implemented using registers only plus a simple control logic.

At the same time, it should be clearly understood that the input and output FIFOs are not a part of the SHA core, but they are rather a part of the surrounding circuit. In particular, during verification of the SHA core using functional, post-synthesis, and timing simulation, these FIFOs can be implemented as a part of a testbench.

Additionally, the inputs and outputs of our proposed SHA core interface do not need to be necessarily generated/consumed by FIFOs. Any circuit that can support control signals `src_ready` and `src_read` can be used as a source of data. Any circuit that can support control signals `dst_ready` and `dst_write` can be used as a destination for data.

The main advantage of using FIFOs is a capability of storing intermediate data awaiting processing. For example, a simple handshaking protocol between the source circuit and the SHA core may tell the source circuit to stop sending data, but it does not support

buffering data that is generated too fast for future processing. This is especially important with sources generating data in a bursting fashion.

3. Format of input

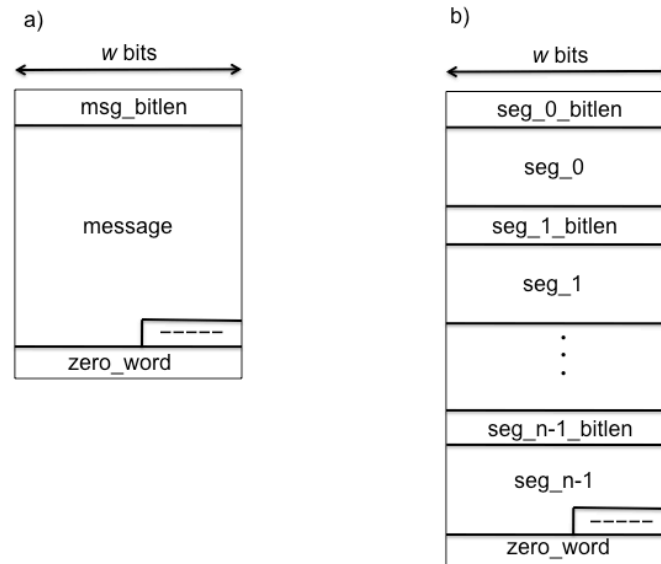


Fig. 3: Format of input data for two different operation scenarios: a) with message bitlength known in advance, and b) with message bitlength unknown in advance. Notation: zero-word = sequence of w zeros, ----- = unused (do not care) bits of the last message word.

The exact format of input to the SHA core is shown in Fig. 3.

Two scenarios of operation are supported. In the first scenario, the message bitlength is known in advance and is smaller than 2^w . In this scenario, shown in Fig. 3a, the first word of input represents message length expressed in bits. This word is followed by all words of the message, including the last word, which can include less than w bits of the message. This last word is followed by the `zero_word`, which indicates the end of the input.

The second format, shown in Fig. 3b, is used when either message length is not known in advance, or it is greater than 2^w . In this case, the message is processed in segments of data denoted as `seg_0`, `seg_1`, ..., `seg_n-1`. For the ease of processing data by the hash core, the size of the segments, from `seg_0` to `seg_n-2` is required to be always an integer multiple of the word size w . The last segment can be of arbitrary length $< 2^w$. This segment is processed in the same way as the entire message in scenario a). This way there is no need for any additional signal to distinguish between these two scenarios. Scenario a) is a special case of scenario b).

Please note that scenario b) is very similar to the way data is assumed to be processed in a typical software API for hash functions, such as [2]. The *Update* function of the software API corresponds to processing segments from `seg_0` to `seg_n-2`. The function *Final*

corresponds to the processing of the last segment of data, `seg_n-1`. The operations of the function *Init* can be either executed when the size of the first block is read, or the `zero_block` of the previous message is processed.

In case the SHA core does not support padding, padding must be done in software, and the meaning of the `msg_bit_length` in Fig. 3a changes to the message bit length after padding. For majority of hash functions known to the authors, this message bit length will be a multiple of the word size, which means that the last word of the message will not contain any unused bits, shown as “do not cares” (-----) in Fig. 3a. Similarly, the sum of the segment bit lengths in Fig. 3b, `seg_0_bitlen + seg_1_bitlen + ... + seg_n-1_bitlen` will have a meaning of the message bit length after padding, and, for the same functions, the last word of the last segment of the message will not contain any unused bits.

Format 3a is a special case of Format 3b. There is no way to distinguish between the two cases, other than by looking at the first word after the end of `seg_0`. If this word is zero, then `seg_0 = entire message`, otherwise, there is at least one more data segment.

Preparing messages in the format shown in Fig. 3b is very simple. The number of clock cycles necessary to process a message smaller than 2^w bits is the same, as in the case of an alternative protocol using two first words of the input to denote the length of the message. For longer messages, any overhead is negligible.

Please note that a single protocol shown in Fig. 3b covers several substantially different scenarios:

- a. message length known or unknown in advance
- b. padding in software or padding in hardware
- c. message length $< 2^w$ or message length $\geq 2^w$.

4. FIFO operation

Each FIFO module is assumed to use synchronous read and synchronous write. In synchronous read, data becomes available at the output of FIFO on the first rising edge of `clk` following an active value of the signal `read`. Similarly, during synchronous write data will be written to the FIFO on the first rising edge of `clk` following the signal `write`. An example of how a FIFO operates after reset is shown below. The assumption is made that when the input `read` is inactive or an attempt is made to read an empty FIFO, the output of the FIFO contains the value “do not care” denoted by “-----“. In the actual implementation, the output of FIFO will contain a specific value, but this value should be ignored by the circuit communicating with the FIFO.

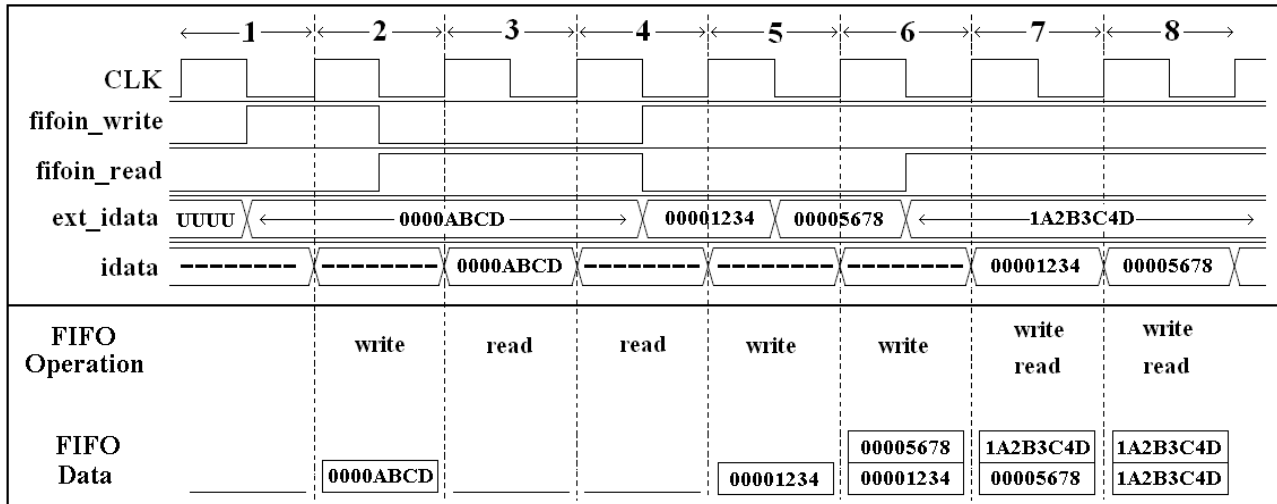


Fig. 4: An example of operation of the Input FIFO.

Literature:

- [1] Federal Information Processing Standard, FIPS 180-3, Secure Hash Standard (SHS), National Institute of Standards and Technology, October 2008, available at http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf
- [2] ANSI C Cryptographic API Profile for SHA-3 Candidate Algorithm Submissions, available at http://csrc.nist.gov/groups/ST/hash/sha-3/Submission_Reqs/crypto_API.html

Contact information:

Kris Gaj (general questions and questions regarding the specification)
 Cryptographic Engineering Electrical and Computer Engineering
 Research Group George Mason University
 email: kgaj@gmu.edu 4400 University Drive
 phone: (703) 993 1575 Fairfax, VA 22030
 fax: (703) 993 1601 U.S.A.