SHA3 Core
Specification

Author: Homer Hsing
homer.hsing@gmail.com

Rev. 0.1
January 29, 2013

This page has been intentionally left blank.

Revision History

Rev.	Date	Author	Description
0.1	01/29/2013	Homer Hsing	First Draft

Contents

1

Introduction

About SHA-3

SHA-3, originally known as Keccak (pronounced [kɛtʃak]) [1], is a cryptographic hash function selected as the winner of the NIST hash function competition [2]. Because of the successful attacks on MD5, SHA-0 and theoretical attacks on SHA-1, NIST perceived a need for an alternative, dissimilar cryptographic hash, which became SHA-3 [3].

SHA-3 is a family of sponge functions that use a permutation as a building block [4]. The permutation (over $\mathbb{Z}_2^{1600}$) is a sequence of operations on a state a that is a three-dimensional array of elements of $GF(2)$, namely $a[5][5][64]$ [4]. The mapping between the bits of $s \in \mathbb{Z}_2^{1600}$ and the bits of a state a is that the $64(5y+x)+z$-th bit of s is $a[x][y][z]$ for any $x \in \mathbb{Z}_5, y \in \mathbb{Z}_5, z \in \mathbb{Z}_{64}$. The permutation consists of 24 rounds, indexed with i_r from 0 to 23. Each round R consists of five steps [4].

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta,$$

$$\theta: \quad a[x][y][z] \leftarrow a[x][y][z] + \sum_{y'=0}^{4} a[x-1][y'][z] + \sum_{y'=0}^{4} a[x+1][y'][z-1],$$

$$\rho: \quad a[x][y][z] \leftarrow a[x][y][z-(t+1)(t+2)/2],$$

with t satisfying $0 \le t < 24$, $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$ in $GF(5)^{2\times 2}$,

or $t = -1$ if $x = y = 0$.

$$\pi: \quad a[x][y] \leftarrow a[x'][y'], \text{with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix},$$

$$\chi: \quad a[x] \leftarrow a[x] + (a[x+1]+1)a[x+2],$$

$$\iota: \quad a \leftarrow a + RC[i_r].$$

$\forall i_r, RC[i_r][x][y] = 0$, if $x \neq 0$ or $y \neq 0$

$\forall i_r, RC[i_r][0][0][2^j - 1] = rc[j + 7i_r], \forall 0 \le j < 6,$

$\forall t, rc[t] = (x^t \bmod x^8 + x^6 + x^5 + x^4 + 1) \bmod x$ in $GF(2)[x]$.

The additions and multiplications are in $GF(2)$.

NIST requires the candidate algorithms to support at least four different output lengths $n \in \{224, 256, 384, 512\}$ with associated security levels [6]. "SHA-3 512", in which output length is 512-bit, has the highest security level among all SHA-3 variants.

Denote the padding of a message M to a sequence of 576-bit blocks by

$$M \parallel pad[576](|M|).$$

The padding is a multi-rate padding, appending a single bit 1 followed by the minimum number of bits 0 followed by a single bit 1 such that the length of the result is a multiple of 576 [4]. Denote the number of blocks of P by $|P|_{576}$, and the $i\text{-}th$ block of P by P_i.

The algorithm of SHA-3 512 is as follows.

Interface: $Z = \text{SHA-3-512}(M), M \in \mathbb{Z}_2^*, Z \in \mathbb{Z}_2^{512}$

$P = M \parallel pad[576](|M|)$
$s = 0^{1600}$
for $i = 0$ to $|P|_{576} - 1$ do
  $s = f\big(s + (P_i \parallel 0^{1024})\big)$
end for
return $\lfloor s \rfloor_{512}$

About this project

This project has implemented "SHA-3 512" hash function.

This project has implemented two cores, one (high-throughput) core designed to work in high clock frequency (150 MHz) dedicated to ASIC or expensive FPGA (Virtex 6), another (low-throughput) core designed to work in low clock frequency (100 MHz) dedicated to cheap FPGA (Spartan 3). Because in many systems the clock frequency is fixed for the entire chip, so even if the hash core can reach a high frequency it has to be clocked at a lower frequency [5].

The code is FPGA-vendor independent, having been fully optimized, using only one clock domain, not using any latch.

2

Architecture

Architecture of the core

The architecture depicted as follows is of the whole core. Two cores implemented in this project have the same architecture.

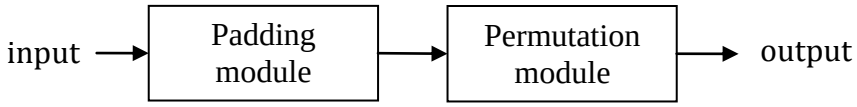


Figure 1: The architecture of the whole core

Architecture of the padding module

The architecture of the padding module is illustrated in the figure below.

The width of the user input is far less than 576 bit. So the padding module uses a buffer to assemble the user input.

If the buffer grows full, the padding module notices the permutation module its output is valid. Then the permutation module begins calculation, the buffer cleared, the padding module waiting for input simultaneously.

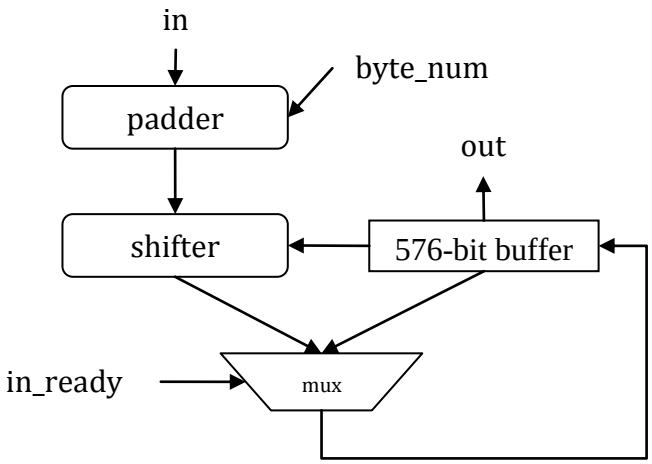


Figure 2: The architecture of the padding module

Architecture of the permutation module

The permutation module of the low throughput core is composed of a combinational logic block computing a round, a counter selecting the round constant, and a register storing the output.

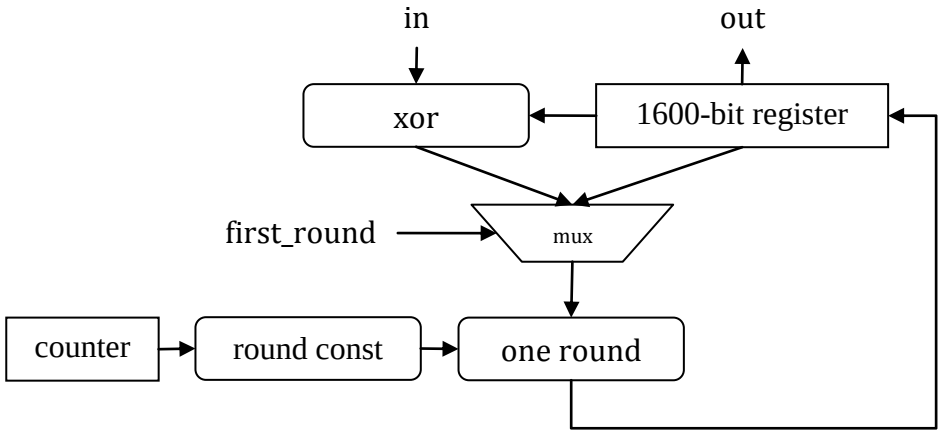


Figure 3A: The architecture of the permutation module (low throughput core)

In the high throughput core, two rounds are done per clock cycle.

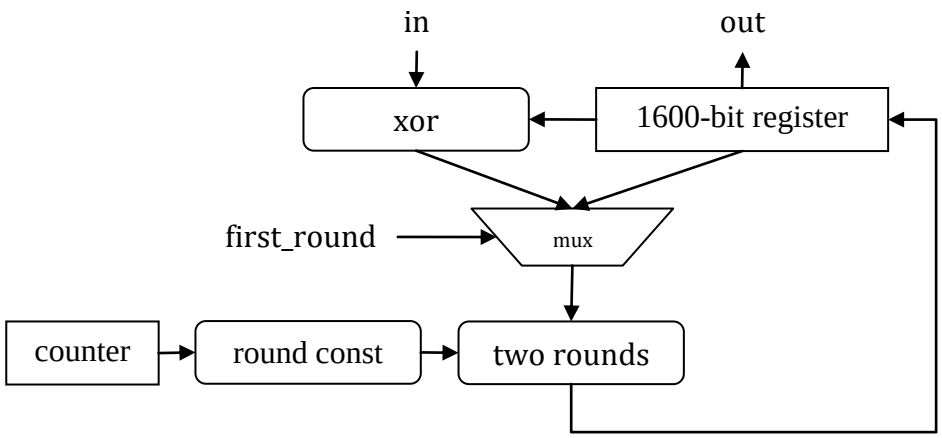


Figure 4B: The architecture of the permutation module (high throughput core)

The round constant module is implemented by combinational logic, saving resource than block RAM, because most bits of the round constant is zero.

3

Ports

The ports and their description are listed in the table below.

Input ports are sampled by the core at the rising edge of the clock.

All output ports are registered, not directly connected to any combinational logic inside the core.

For ports wider than 1 bit, its bit index is from the port width minus one down to zero.

Table 1: Ports of the high throughput core

Port name	Width	Direction	Description
clk	1	In	Clock
reset	1	In	Synchronous reset
in	64	In	Input
byte_num	3	In	The byte length of *in*
in_ready	1	In	Input is valid or not
is_last	1	In	Current input is the last or not
buffer_full	1	Out	Buffer is full or not
out	512	Out	The hash result
out_ready	1	Out	The hash result is ready or not

Table 2: Ports of the low throughput core

Port name	Width	Direction	Description
clk	1	In	Clock
reset	1	In	Synchronous reset
in	32	In	Input
byte_num	2	In	The byte length of *in*
in_ready	1	In	Input is valid or not
is_last	1	In	Current input is the last or not
buffer_full	1	Out	Buffer is full or not
out	512	Out	The hash result
out_ready	1	Out	The hash result is ready or not

The width of port *in* and port *byte_num* is difference between the high throughput core and the low throughput core. The reason is we want to get high performance. When the permutation module is computing by the current input, the padding module is preparing next input. We don't want the permutation wait the padding. For the high throughput core, permutation is done in 12 clock cycles, so its padding module should prepare next 576 bit in time, so its input width is 64 bit. For the low throughput core, which finishes permutation in 24 clock cycles, the input width is fixed in the same way.

4

Usage and timing

Before computing a new hash value each time, reset the core. Suppose the core has computed SHA-3-512("A"). Then you should first reset the core before letting it compute SHA-3-512("B"). The reset method is to let $reset$ be 1, then after one clock cycle to let $reset$ be 0. The rising edge of $reset$ should be aligned to the rising edge of clk.

If real width of the input is less than the width of input port, in should be aligned to the most significant bit. This may only happens at the last block of input. Suppose the input is one byte data "A". You should let $in[63:56]$="A", ($in[31:24]$ for the low throughput core) not $in[7:0]$="A".

If and only if current input is the last block, is_last is 1. The value of $byte\_num$ takes effect only when is_last is 1. If the message length is multiple of the input port width, you should provide an additional zero-length block. For example, to hash "ABCDEFGH", first set

$$in = "ABCDEFGH", is\_last = 0$$

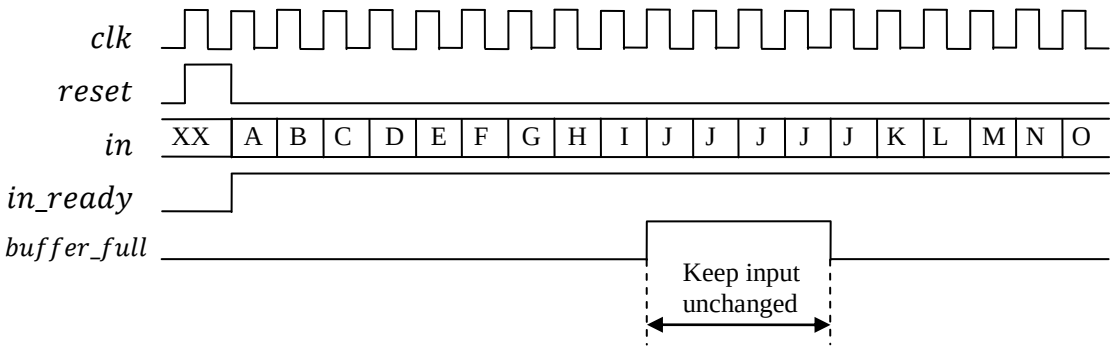
Then set

$$is\_last = 1, byte\_num = 0$$

You may provide input in any slow rate. If input is not ready, just let in_ready be 0, then the core will not absorb current input value.

Only if $buffer\_full$ is 0 and in_ready is 1, the core absorbs current value. If in any clock cycle $in = x, in\_ready = 1, buffer\_full = 1$ then next valid value of in should be x. A timing graph below helps understand it.

The hash result is ready only if out_ready is 1. After out_ready is 1, the hash result remain unchanged until next reset.


clk
reset
in
XX
A
B
C
D
E
F
G
H
I
J
J
J
J
J
K
L
M
N
O
in_ready
buffer_full
Keep input
unchanged

5

Synthesis result

Synthesis result

The synthesis software is Xilinx ISE version 14.4.

The high throughput core has been synthesized targeting an expensive Virtex 6 FPGA.

Table 3: synthesis result of the high throughput core

Device	Xilinx Virtex 6 XC6VLX240T-1FF1156
Number of Slice Registers	2,220
Number of Slice LUTs	9,895
Number of fully used LUT-FF pairs	1,673
Number of bonded IOBs	585
Number of BUFG/BUFGCTRLs	1
Maximum Frequency	188.9 MHz

The low throughput core has been synthesized targeting a very cheap Spartan 3 FPGA.

Table 4: synthesis result of the low throughput core

Device	Xilinx Spartan 3 xc3s5000-4fg900
Number of Slices	2,321
Number of Slice Flip Flops	2,346
Number of 4 input LUTs	4,499
Number of bonded IOBs	552
Number of GCLKs	1
Maximum Frequency	117.3 MHz

Throughput

For the high throughput core: 7.2 G bit /second if clock frequency is 150 MHz

For the low throughput core: 2.4 G bit /second if clock frequency is 100 MHz

6

Testbench

The RTL files and test benches for the high throughput core are in directory "high_throughput_core". The files for the low throughput core are in directory "low_throughput_core". The structure of two directories is same.

The file "testbench/simulation.do" is a batch file for ModelSim to compile the HDL files, setup the wave file, and begin function simulation. In order to make it work properly, the working directory of ModelSim must be the directory of "testbench".

The file "testbench/test_keccak.v" is the main test bench for the cores. The test feeds input data to the core and compares the correct result with the core output. For any wrong output, the test bench displays error messages. If all output being correct, the test bench displays "OK".

7

References

[1] Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche, "The Keccak sponge function family: Specifications summary", http://keccak.noekeon.org/specs_summary.html

[2] "NIST Selects Winner of Secure Hash Algorithm (SHA-3) Competition", NIST. Oct. 2012. http://www.nist.gov/itl/csd/sha-100212.cfm

[3] "SHA-3", Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/SHA3

[4] The Keccak reference, version 3.0, http://keccak.noekeon.org/Keccak-reference-3.0.pdf

[5] Keccak implementation overview, version 3.2, http://keccak.noekeon.org/Keccak-implementation-3.2.pdf

[6] "Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family", Federal Register Notices 72 (2007), no. 212, 62212–62220 http://csrc.nist.gov/groups/ST/hash/index.html

[7] K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif, "Comprehensive evaluation of high-speed and medium-speed implementations of Five SHA-3 Finalists using Xilinx and Altera FPGAs", 3rd SHA-3 candidate conference, Mar 2012.