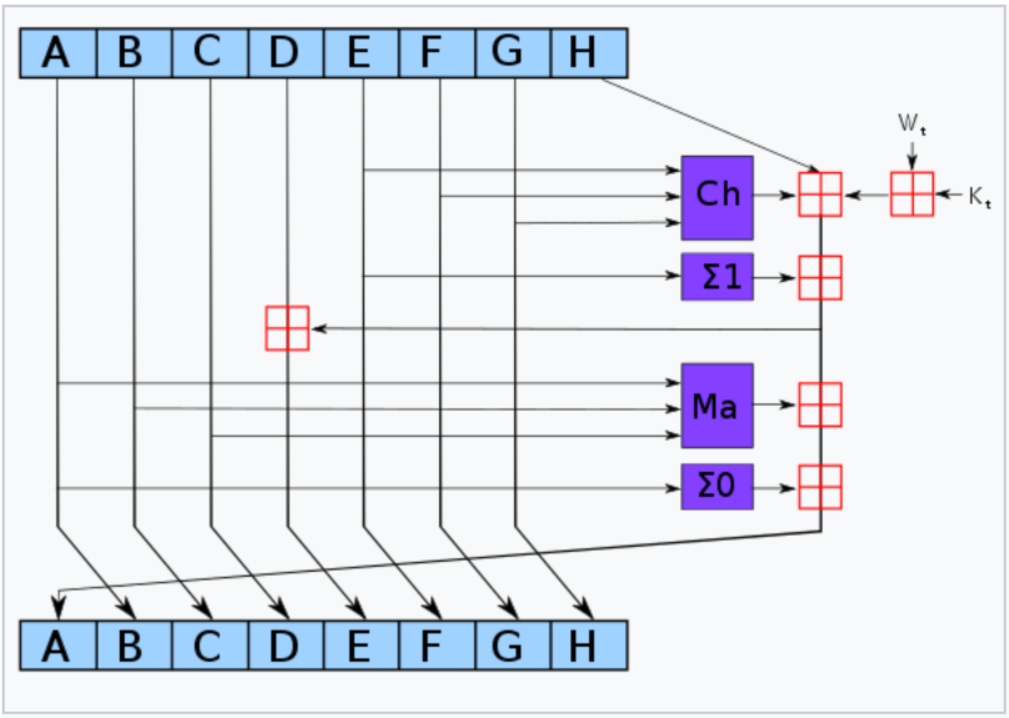
1. Specification

The aim of the project is to design a SHA256 hardware accelerator that will be synthesized for and mapped on the Zynq core of the Zybo board by Digilent. The hardware block will have an AXI compliant wrapper which allows it to take inputs and write outputs on specific registers using the AXI protocol.

SHA-2 (Secure Hash Algorithm 2) is a set of cryptographic hash functions designed by the United States National Security Agency (NSA) and first published in 2001. They are built using the Merkle–Damgård construction, from a one-way compression function itself built using the Davies–Meyer structure from a specialized block cipher. [1][2]



One iteration in a SHA-2 family compression function. The blue components perform the following operations:

$$\mathrm{Ch}(E,F,G) = (E \land F) \oplus (\neg E \land G)$$
$$\mathrm{Ma}(A,B,C) = (A \land B) \oplus (A \land C) \oplus (B \land C)$$
$$\Sigma_0(A) = (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22)$$
$$\Sigma_1(E) = (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25)$$

The bitwise rotation uses different constants for SHA-512. The given numbers are for SHA-256.

The red ⊞ is addition modulo 2^{32} for SHA-256, or 2^{64} for SHA-512.

Image 1. SHA256 operations [1]

[1] https://en.wikipedia.org/wiki/SHA-2

1.1 Operations on Words

For SHA256, each message block has **512 bits,** which are represented as a sequence of sixteen 32-bit words.

1. Bitwise logical word operations.
2. Addition modulo 2^w
3. The *right shift* operator **SHR^n(x).**
4. The rotate right (circular right shift) operation **ROTR^n(x).**
5. The rotate left (circular left shift) operation, **ROTL^n(x).**

1.2 SHA-256 Functions

SHA-256 uses six logical functions, where each function operates on 32-bit words, which are represented as x, y, and z. The result of each function is a new 32-bit word.

$$\begin{aligned} Ch(x,y,z) &= (x \wedge y) \oplus (\quad x \wedge z) \\ Maj(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \end{aligned}$$

$$\begin{aligned} \textstyle\sum_0^{\{256\}}(x) &= ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \\ \textstyle\sum_1^{\{256\}}(x) &= ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \\ \sigma_0^{\{256\}}(x) &= ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \\ \sigma_1^{\{256\}}(x) &= ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x) \end{aligned}$$

Image 2. SHA-256 Functions [2]

[2] http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

2. Profiling

Profiling of the SHA-256 algorithm was done using the valgrind tool.

https://developer.mantidproject.org/ProfilingWithValgrind.html

The results are represented in the table bellow.

Function name	Incl.	Self
pad()	24.79%	0.03%
hash()	11.62%	3.62%
printHash()	8.71%	0.07%

Where the parameters are the following:

Incl. - Sum of itself + all child calls as a percentage of the whole. Programs with little static allocation should have main() at 100%. Units are those selected by the to-right drop-down

Self - Exclusive count spent in the selected function. Units are those selected by the to-right drop-down

The pad() function requires an input function paramater from the user (a string to be hashed). That is why it requires more processing time compared to the hash() and printHash() functions.

The user interface prompts the user to provide a message to hash. Since the message will be inserted through the keyboard it will always be a multiple of 8 bits.

The first part of the SHA256 is the padding, which can be summarized in three operations:

- append a '1' bit at the end of the message
- append K '0' bits, where K is the minimum number >= 0 such that Len(message) + 1 + K + 64 is a multiple of 512
- append the length of the message in the last 64 bits

These operations are not computationally complex and, in addition, they are not regular (in the worst case a new 512-bit block must be added). **Therefore, the padding will be implemented in software by the application itself. The hash() function will be hardware accelerated.**

3. Bit analysis

This chapter represents the results of the bit analysis. The hash() function, which will be accelerated, uses only integer values, therefore a static bit analysis is required.

The SHA-256 algorithm has the input message size < 2^{64}-bits. Block size is 512-bits, and it has a word size of 32-bits. The output of the hash algorithm is 256 bits (32 bytes).

According to these constraints, the analysis was done for the variables used in the hash function. The variables are v, W, H, M, T1 and T2.

The padded message is parsed into N 512-bit blocks, $M^{(1)}$, $M^{(2)}$,...,$M^{(N)}$. Since the 512 bits of the input block may be expressed as sixteen 32-bit words, the first 32 bits of message block *i* are denoted $M_0^{(i)}$, the next 32 bits are $M_1^{(i)}$, and so on up to $M_{15}^{(i)}$.

After repeating multiple steps of the hash function a total of N times (i.e., after processing $M^{(N)}$), the resulting 256-bit message digest of the message, M, is:

$$H_0^{(N)} \,||\, H_1^{(N)} \,||\, H_2^{(N)} \,||\, H_3^{(N)} \,||\, H_4^{(N)} \,||\, H_5^{(N)} \,||\, H_6^{(N)} \,||\, H_7^{(N)} .$$

All of the variables inside the hash function are positive integer numbers, after the static analysis, all of the mentioned variables must be 32 bits.