IOb-SoC-OpenCryptoHW Acceleration Plan

OpenCryptoHW Project Report

IObundle

September 14, 2022

Contents

List of Tables

List of Figures

1 Introduction

A secure and trustworthy internet needs computationally intensive cryptography algorithms. For security, performance and applicability to multiple platforms, it is best if these algorithms are executed in hardware with some of the flexibility of software.

The OpenCryptoHW project, funded by the NGI Assure program, proposed the use of Coarse Grained Reconfigurable Arrays (CGRAs) [1] to accomplish this objective, and chose the SHA256 and the AES256 algorithms to illustrate the ideas proposed.

The project's assumptions are the following: internet devices use a main Central Processing Unit and an Ethernet Medium Access Controller (MAC) core. Each internet device is engaged in frequent secure network communications. The software and hardware descriptions must be open-source to ensure trust. OpenCryptoHW uses the VexRiscV processor [2] and the IOb-Eth MAC core [3] to accomplish these objectives. Moreover, it uses the IOb-Versat CGRA [4] to accelerate the cryptography algorithms.

This document describes the SHA256 [5] acceleration plan running on the IOb-SoC-SHA system [6], according to Milestone 7 of the OpenCryptoHW project. The document has five parts:

- the first part provides a brief introduction to the SHA-256 algorithm.
- the second part introduces the IOb-Versat framework and related tools.
- the third part presents profiling data for the SHA256 application running exclusively on the RISC-V CPU.
- the fourth part elaborates on the profiling conclusions and establishes functional unit architectures to accelerate the application.
- the fifth part outlines a prediction for the expected results from implementing the acceleration strategy.

2 SHA256 Algorithm

The SHA-256 algorithm [5] is a secure hash algorithm that receives input data of any size up to 2^{64} bits and computes an output of 256 bits. This output is called a message digest. The SHA-256 algorithm has two main stages: preprocessing and hash computation.

2.1 Preprocessing

The preprocessing stage pads the input data to obtain an input size multiple of 512 bits. Given a message M of size λ bits. The padding process appends the bit "1" to the end of the message, followed by δ "0" bits such that δ is the smallest positive integer that solves (1).

$$\lambda + 1 + \delta \equiv 448 \mod 512. \tag{1}$$

After the padded zeroes, the message is appended with the 64-bit representation of the size of the original message λ. At the end of this process, the padded message size is a multiple of 512 bits. The padded input splits into blocks of 512 bits. Each blocks forms a set of sixteen words of 32 bits.

The preprocessing stage also sets the initial state for the hash value. The hash state values are a set of eight 32 bit words. For the SHA-256 algorithm, the initial values are the first 32 bits of the fractional part of the square root of the first eight prime numbers.

2.2 Hash Computation

Figure 1 presents the hash computation stage which processes one message block at a time. For each iteration i, the hash stage values $H_0^{(i+1)}, H_1^{(i+1)}, ..., H_7^{(i+1)}$ are updated using a message schedule of sixty-four 32 bit words $W_0, W_1, ..., W_{63}$, the previous hash state values $H_0^{(i)}, H_1^{(i)}, ..., H_7^{(i)}$ and 64 constants $K_0, K_1, ..., K_{63}$ of 32 bit each.

Figure 1: SHA-256 hash function flowchart.

The initial hash state values and the message block words come from the preprocessing stage as described in section 2.1. The 64 constants K_t are the 32 fractional bits of the cubic roots of the first 64 prime numbers.

The sixty-four message schedule words are the sixteen 32 bit words from the input message block plus 48 generated words. Each generated word W_t is computed by the operations presented in (2). The addition is modulo 2^{32}.

$$W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}, \quad 16 \leq t \leq 63. \tag{2}$$

Where $\sigma_0()$ and $\sigma_1()$ functions are a set of logic operations defined in (3). $ROTR^n(x)$ is a rotate right n bits function and $SHR^n(x)$ is a right shift n bits operation.

$$\sigma_0(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x),$$
$$\sigma_1(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x). \tag{3}$$

The initial hash state values initialize a set of working variables a, b, c, d, e, f, g, h:

$$\begin{aligned} a &= H_0^{(i-1)} \\ b &= H_1^{(i-1)} \\ c &= H_2^{(i-1)} \\ d &= H_3^{(i-1)} \\ e &= H_4^{(i-1)} \\ f &= H_5^{(i-1)} \\ g &= H_6^{(i-1)} \\ h &= H_7^{(i-1)}. \end{aligned} \tag{4}$$

The working variables are updated for 64 iterations ($0 \leq t \leq 63$), following the algorithm in (5). The functions $\Sigma_1(x)$, $Ch(x, y, z)$, $\Sigma_0(x)$ and $Maj(x, y, z)$ are defined in (6). $Ch(x, y, z)$ is a choice operation: if x is 1, the output is z, otherwise outputs y. $Maj(x, y, z)$ outputs the most common value between the three inputs.

$$\begin{aligned} T_1 &= h + \Sigma_1(e) + Ch(e, f, g) + K_t + W_t \\ T_2 &= h + \Sigma_0(a) + Maj(a, b, c) \\ h &= g \\ g &= f \\ f &= e \\ e &= d + T_1 \\ d &= c \\ c &= b \\ b &= a \\ a &= T_1 + T_2. \end{aligned} \tag{5}$$

$$\begin{aligned} \Sigma_0(x) &= ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x), \\ \Sigma_1(x) &= ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x), \\ Ch(x, y, z) &= (x \wedge y) \oplus (\neg x \wedge z), \\ Maj(x, y, z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z). \end{aligned} \tag{6}$$

The results $H_0^{(i+1)}, H_1^{(i+1)}, ..., H_7^{(i+1)}$ of hashing iteration i are the final values of the working variables $a, ..., h$. The resulting hash state values of one iteration are the input of the next. This process repeats for all message blocks. The hash state values at the final iteration concatenated form the message digest.

3 IOb-Versat CGRA

The IOb-Versat [4] CGRA aims to provide a framework and a set of tools to facilitate the development of CGRA architectures. A CGRA is a set of multiple interconnected functional units (FUs). A CPU can configure the FUs and the connections between them to change the functionality and data flow of the hardware architecture at runtime.

For a particular application, the developer needs to establish the required FUs. The developer can choose from a set of default FUs supported by IOb-Versat. These include external memory access, addition, multiplication and accumulation, internal memory blocks, simple arithmetic and logic unit and multiplexing, among other FUs.

IOb-Versat also supports custom designed FUs in Verilog. The integration with IOb-Versat requires the development of corresponding FU wrappers in C/C++ to allow simulation and software driver control.

The computation datapaths are described in C/C++ by interconnecting FU wrappers. IOb-Versat provides tools to automatically synchronize the data along the datapaths and merge multiple computation datapaths. The datapaths are synchronized using the delays of each FU type and the datapath graph. For cases where different datapaths have sets of FUs in common, the IOb-Versat can merge the computation datapaths to reduce the hardware footprint.

IOb-Versat drivers that configure the CGRA and execute the datapath replace the software routines. The IOb-Versat accesses the system memory directly through a direct memory access (DMA) block, thus bypassing the CPU+cache subsystem. Figure 2 presents an IOb-SoC system with Versat CGRA as a peripheral. The CPU accesses the Versat CGRA configuration values, sends run commands or reads status.

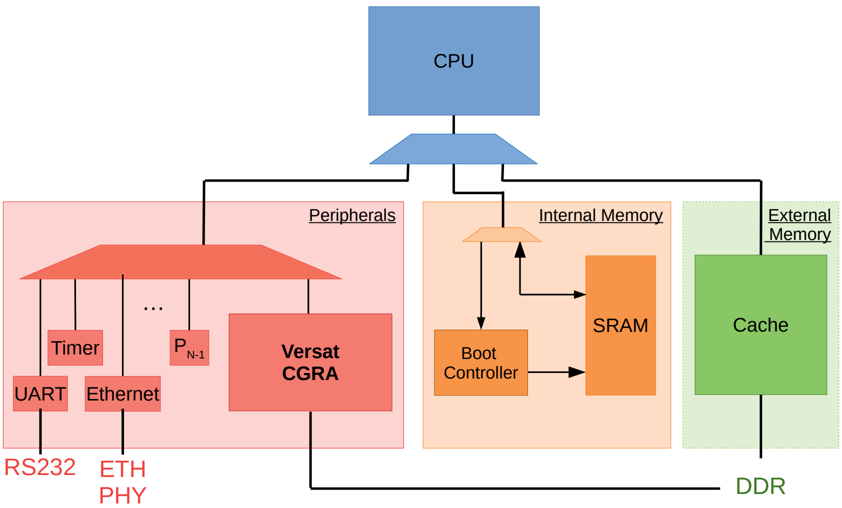

Figure 2: Example of Versat CGRA integration in IOb-SoC.

Figure 3 provides a block diagram of a Versat CGRA. The CPU accesses affect the configuration block. The configuration block fans out the configuration values for each specific FU. Versat supports any topology between the FUs in the configured datapaths. The **vRead** and **vWrite** are specified in the block diagram to highlight the interface with external memory.

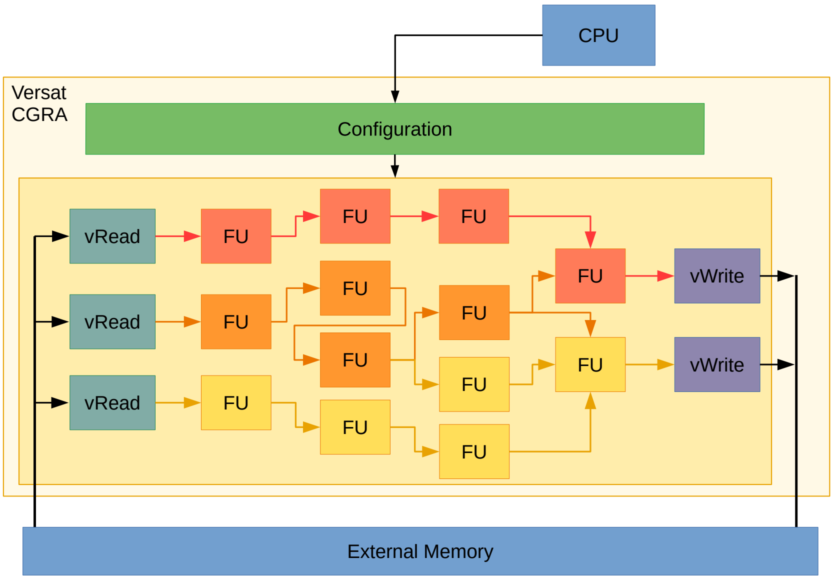

Figure 3: Example of Versat CGRA block diagram.

4 Profiling Assessment

The execution time for the SHA256 implementation [6] is presented in Table 1. The program runs exclusively on software with instructions stored in internal memory and data stored in external memory (DDR). The system uses the VexRiscv CPU [7] at a clock frequency of 100 MHz.

Function	Time (μs)	Time (%)
Global	28381	100
sha256	25360	89
sha_init	886	3
sha_finalize	22816	80
crypto_hashblocks	19435	68
ld_big_endian	2262	7
st_big_endian	1282	4
F_32	4686	16
Expand32	2755	9
sha_ctxrelease	571	2
mem	1351	4

Table 1: Baseline application profile data.

The profile analysis tracks the time in clock cycles since the input data is in external memory until the CPU writes the output data to external memory.

The results from Table 1 demonstrate that about 80% of the execution time is used to run the `sha_finalize()` function, in particular, the `crypto_hashblocks()` function. The functions and macro calls inside the `crypto_hashblocks()` function have the same order of magnitude with regards to duration.

The acceleration efforts should be focussed on the `crypto_hashblocks()` function and respective subfunction and macro calls.

5 Acceleration Proposal

The `crypto_hashblocks()` function has a similar flowchart to the generic SHA-256 algorithm presented in Figure 1. The function starts by reading the current hash state values from memory. Then uses each message block of 64 bytes (512 bits) of the input data to hash the message block. After using all input data, the function writes the new state to memory. Most computations take place inside the loop to hash the message blocks.

Figure 4 presents a block diagram for the process of hashing a message block. The message block hashing output is the accumulation from the initial state with the newly computed state. The newly computed state is the output of the sequence of **F** blocks. Each **F** block receives three inputs: a set of constants stored in the **cMem** blocks; the previous or initial state in the **a-h** variables; and a set of words from the message scheduling array.

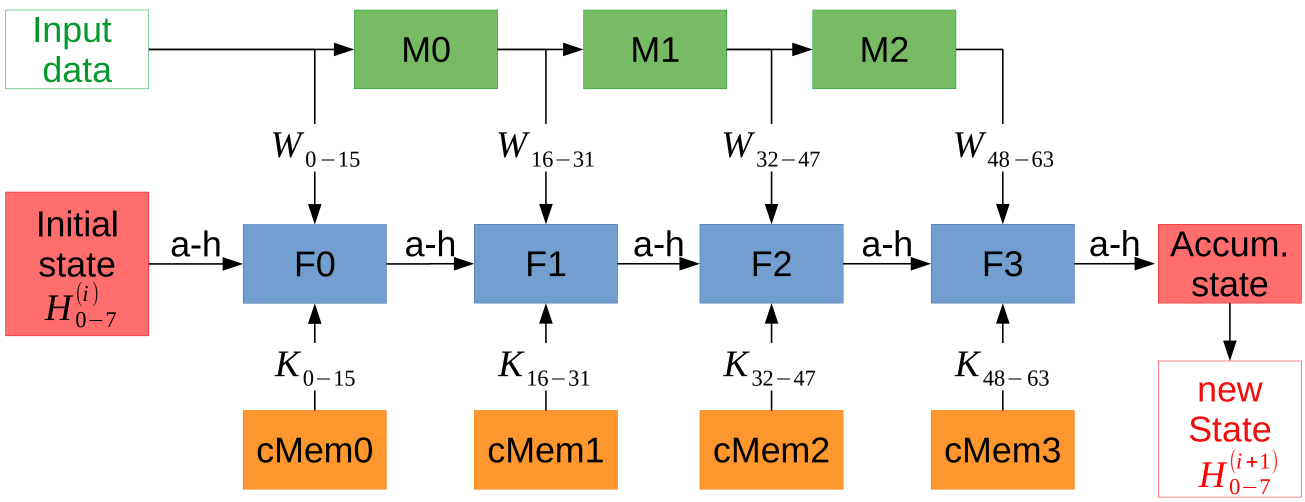

Figure 4: Hash message block block diagram.

Each set of 16 W words is obtained from the input data or by applying a previous set of words to the **M** block.

The proposed accelerator architecture has 5 functional unit (FU) types:

- 1 Vread to store the input data;
- 1 State FU to store and accumulate the **a-h** state variables;
- 3 Memories to store the constants (equivalent to **cMem** blocks);
- 3 **M** FUs that generate a new set of message schedule array words;
- 4 **F** FUs that perform the compression function.

The Vread, state and memory FUs are default FUs from Versat. The **M** and **F** FUs are custom units built specifically for the SHA256 application.

The Vread FU reads data from any memory address in the system and provides input data to other FUs.

The memory FU is an auxiliary memory that holds constant values used as input to other FUs.

The state FU is a set of accumulation registers. The registers can be initialized with a value by the CPU. The register values can be inputs or outputs of other FUs.

The **M** FU performs the logic equivalent to the `EXPAND_32` macro defined in the `sha.c` source code. The macro generates 16 new words for the message schedule array following (2).

The **F** FU performs the logic equivalent to a group of 16 `F_32(w,k)` macros defined in the `sha.c` source code. Each `F_32(w,k)` macro updates the state values following the expressions in (5).

6 Expected Results

The best-case performance for the acceleration of a hash function is to compute the working variables equations (5) once per cycle. This case gives 64 clock cycles to hash a message block. The test messages used in the profiling application have increasing sizes from 0 to 512 bits with a step size of 8 bits. The test messages with sizes from 0 to 440 bits are split into one message block. The remaining test messages require two message blocks. The minimum number of clock cycles to execute the hashing of message blocks for the complete test is given in (7). The test contains 65 messages with the last 9 split into two message blocks during hashing.

$$\begin{aligned} hash\_cycles = 64 \times (65 + 9) = 4736 \; clock \; cycles \\ 4736 \; clock \; cycles \Leftrightarrow \frac{4736}{100 \times 10^6} \times 10^6 = 47 \; \mu s. \end{aligned} \tag{7}$$

The acceleration proposed in section 5 replaces the `sha_init()`, `sha_finalize()` and `sha_ctxrelease()` functions. From table 1, these functions take a total of $(886 + 22816 + 571) = 24273 \; \mu s$ to execute. Assuming that the SHA-256 acceleration takes the time calculated in (7), the expected speedup is given by:

$$\frac{Total \; time}{Accel \; time} = \frac{28381}{28381 - 24273 + 47} \approx 6.83. \tag{8}$$

7 Conclusion

This document presents the SHA256 [5] acceleration plan running on the IOb-SoC-OpenCryptoHW system [**?**], according to Milestone 7 of the OpenCryptoHW project.

The IOb-SoC-OpenCryptoHW system consists of a VexRiscV processor [2], the IOb-Eth MAC core [3], and the IOb-Versat CGRA [4] to accelerate the cryptography algorithms.

First, the document briefly introduces the sha256 cryptographic algorithm. Then the execution profile of a pure software implementation runs on the Vex RiscV CPU is derived and discussed.

Based on the profile results, a hardware accelerator using the Versat CGRA is proposed. The CGRA can morph into a different acceleration to support other algorithms. The proposed system is analysed and expected acceleration of almost seven times compared to software is derived.

This non-FPGA reconfigurable hardware implementation not only improves performance but also increases security and decreases energy consumption.

References

[1] M. Wijtvliet, L. Waeijen, and H. Corporaal. Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 235–244, July 2016.

[2] SpinalHDL. Vexriscv. `https://github.com/SpinalHDL/VexRiscv`, 2022.

[3] IObundle Lda. Iob-eth. `https://github.com/IObundle/iob-eth`, 2022.

[4] IObundle Lda. Iob-versat. `https://github.com/IObundle/iob-versat`, 2022.

[5] Quynh Dang. Secure hash standard, 2015-08-04 2015.

[6] IObundle Lda. Iob-soc-opencryptohw. `https://github.com/IObundle/iob-soc-opencryptohw`, 2022.

[7] IObundle Lda. Iob-vexriscv. `https://github.com/IObundle/iob-vexriscv`, 2022.

