IOb-SoC-SHA

Acceleration Proposal

April 27, 2022

Contents

List of Tables

List of Figures

1 Introduction

This document outlines an acceleration plan for the SHA256 application execution in the IOb-SoC-SHA system.

The document is divided in two parts:

- the first part presents profiling data for the SHA256 application running exclusively on the riscv CPU.
- the second part elaborates on the profiling conclusions and establishes functional unit architectures to accelerate the application.

2 Profiling Assessment

The execution time for the SHA256 application 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 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 the external memory, until the output data in stored in the 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.

3 Acceleration Proposal

Figure 1 presents the flowchart of the `crypto_hashblocks()` function.

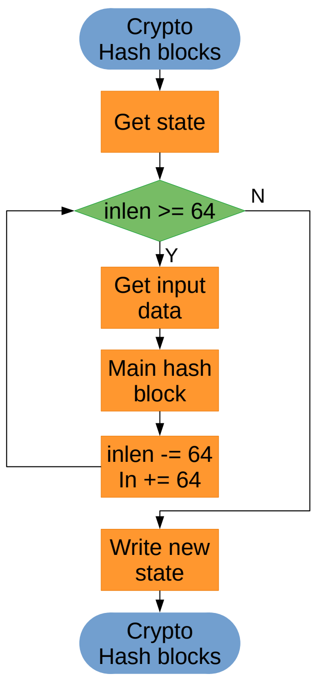

Figure 1: `crypto_hashblocks()` function flowchart.

The function starts by reading the current state from memory. Then each block of 64 bytes (256 bits) of the input data is used as input to the main hash block. After all input data is used, the new state is written to memory. The majority of computations take place in the main hash block inside the loop.

Figure 2 presents a diagram for the main hash block.

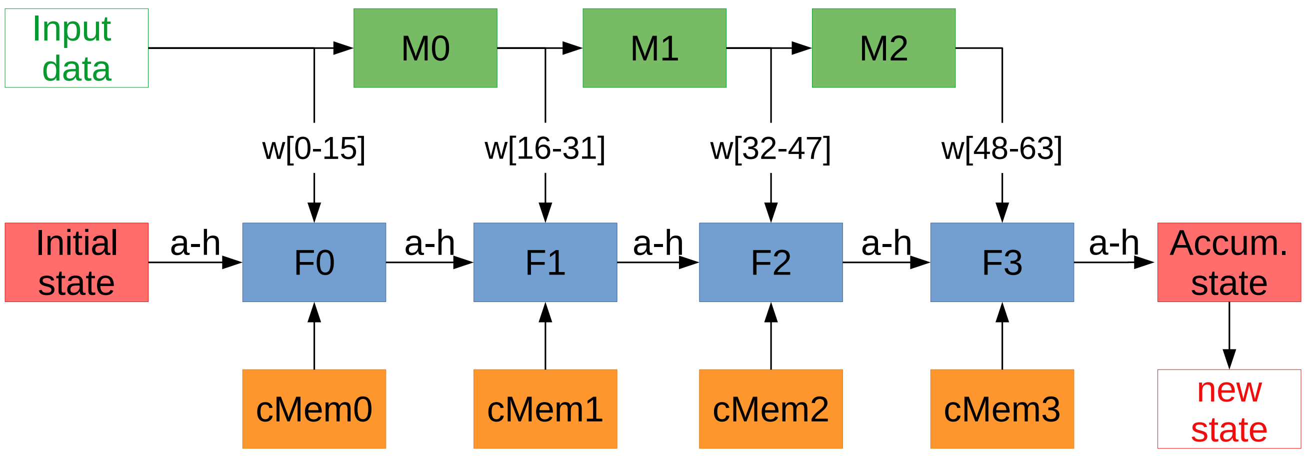

Figure 2: Main hash block diagram.

The main hash block output is the accumulation from the initial state with the new computed state. The new 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 **w**.

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 **M** FU performs the logic equivalent to the `EXPAND_32` macro presented in Listing 1. The macro generates 16 new words for the message schedule array. Each new word is generated by applying logic operations to previous words.

```
#define SHR(x, c) ((x) >> (c))
#define ROTR_32(x, c) (((x) >> (c)) | ((x) << (32 - (c))))
#define ROTR_64(x, c) (((x) >> (c)) | ((x) << (64 - (c))))

#define sigma0_32(x) (ROTR_32(x, 7) ^ ROTR_32(x,18) ^ SHR(x, 3))
#define sigma1_32(x) (ROTR_32(x,17) ^ ROTR_32(x,19) ^ SHR(x,10))

#define M_32(w0, w14, w9, w1) w0 = sigma1_32(w14) + (w9) + sigma0_32(w1) + (w0);

#define EXPAND_32           \
    M_32(w0, w14, w9, w1)   \
    M_32(w1, w15, w10, w2)  \
    M_32(w2, w0, w11, w3)   \
    M_32(w3, w1, w12, w4)   \
    M_32(w4, w2, w13, w5)   \
    M_32(w5, w3, w14, w6)   \
    M_32(w6, w4, w15, w7)   \
    M_32(w7, w5, w0, w8)    \
    M_32(w8, w6, w1, w9)    \
    M_32(w9, w7, w2, w10)   \
    M_32(w10, w8, w3, w11)  \
    M_32(w11, w9, w4, w12)  \
    M_32(w12, w10, w5, w13) \
    M_32(w13, w11, w6, w14) \
    M_32(w14, w12, w7, w15) \
    M_32(w15, w13, w8, w0)
```

Listing 1: `EXPAND_32` macro.

The **F** FU performs the logic equivalent to a group of 16 `F_32(w,k)` macros which are presented in Listing 2. Each `F_32(w,k)` macro updates the state values (**a** to **h**) using one message schedule word, one constant value and the previous state values.

```
#define Ch(x, y, z) (((x) & (y)) ^ (~(x) & (z)))
#define Maj(x, y, z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))

#define Sigma0_32(x) (ROTR_32(x, 2) ^ ROTR_32(x,13) ^ ROTR_32(x,22))
#define Sigma1_32(x) (ROTR_32(x, 6) ^ ROTR_32(x,11) ^ ROTR_32(x,25))

#define F_32(w, k)                                        \
    T1 = h + Sigma1_32(e) + Ch(e, f, g) + (k) + (w);     \
    T2 = Sigma0_32(a) + Maj(a, b, c);                     \
    h = g;                                                \
    g = f;                                                \
    f = e;                                                \
    e = d + T1;                                           \
    d = c;                                                \
    c = b;                                                \
    b = a;                                                \
    a = T1 + T2;


F_32(w0, const0)
F_32(w1, const1)
F_32(w2, const2)
F_32(w3, const3)
F_32(w4, const4)
F_32(w5, const5)
F_32(w6, const6)
F_32(w7, const7)
F_32(w8, const8)
F_32(w9, const9)
F_32(w10, const10)
F_32(w11, const11)
F_32(w12, const12)
F_32(w13, const13)
F_32(w14, const14)
F_32(w15, const15)
```

Listing 2: Group of 16 `F_32(w,k)` macros.