RTES - Report Final Project

Gianluca Radi, Saverio Nasturzio

June 5, 2022

Contents

1 Introduction

The aim of this project is to simulate the functioning of a block chain that employs a simplified version of the proof-of-work technique[1], in a game-like manner. In particular, we have designed hardware accelerators (**hashers**) that, given an array of 512-bits blocks, compute the hashes for each of the input block using the Secure Hashing Algorithm 1 (SHA-1)[2]. Additionally, each of the resulting hashes should have a certain number of 0s as the most significant bits. This number of 0s is called **complexity**. Then, the hashes must be written in the memory. The game is divided into rounds of increasing difficulty and the hasher that has finished first for most of the rounds will be the winner of the game. Moreover, some profiling is executed to compare the performances of a fully hardware approach, a custom instruction-aided software approach, and a pure software approach to compute the correct hashes of the blocks.

2 System

For the project, we developed the system shown in Figure 1. The Top-Level system is composed of:

- **NIOS II**: runs the application.
- **JTAG UART**: used for debugging purposes.
- **On-Chip Memory**: stores the instructions and the data (input blocks).
- **8-bit Bidirectional PIO**: used for profiling
- **SHA1Hasher_0**: accelerator for the computation of the hashes.
- **SHA1Hasher_1**: accelerator for the computation of the hashes.
- **SHA1Hasher_2**: accelerator for the computation of the hashes.
- **SHA1Hasher_3**: accelerator for the computation of the hashes.

It is worth noting that more configurations were tried, but unfortunately no more than 4 hashers can be instantiated due to resource limitation on the board.

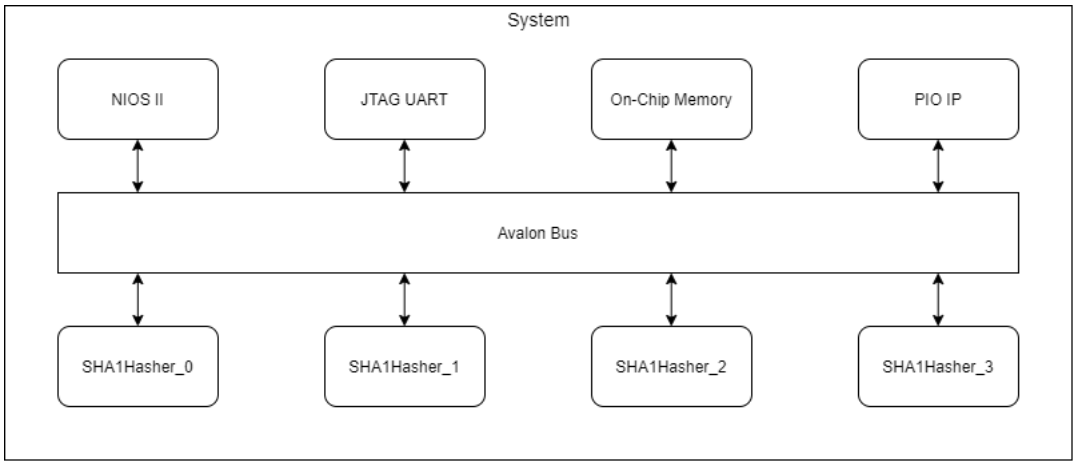

Figure 1: Representation of the top level system.

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

3 SHA1Hasher

In this section, we will discuss the only custom hardware IP that is employed in this lab: the SHA1Hasher[3].
The objective of this IP is to compute the correct hashes for each of the input blocks stored in the memory and to write them back starting from a specified address. Note that with *correct hash* we mean a hash compliant with the specified complexity (i.e. having as the most significant bits the specified number of '0's).
As figure 2 illustrates, the SHA1Hasher is comprised of 4 sub-modules that carry out different functions. Firstly, we need an Avalon Slave interface to allow the user to program the accelerator. Secondly, an Avalon Master must be provided as the accelerator needs to fetch the input blocks from the memory and write back the computed hashes. In this sense, the Master sub-module simply implements a DMA. Thirdly, we need a component that implements the SHA1 algorithm. That is, given as input a block of 512 bits, it needs to return the SHA1 hash for that block. Finally, a Controller is needed to coordinate the functioning of the other sub-modules. The Controller also provides a Conduit interface. In the following subsections we will analyze each of these components.

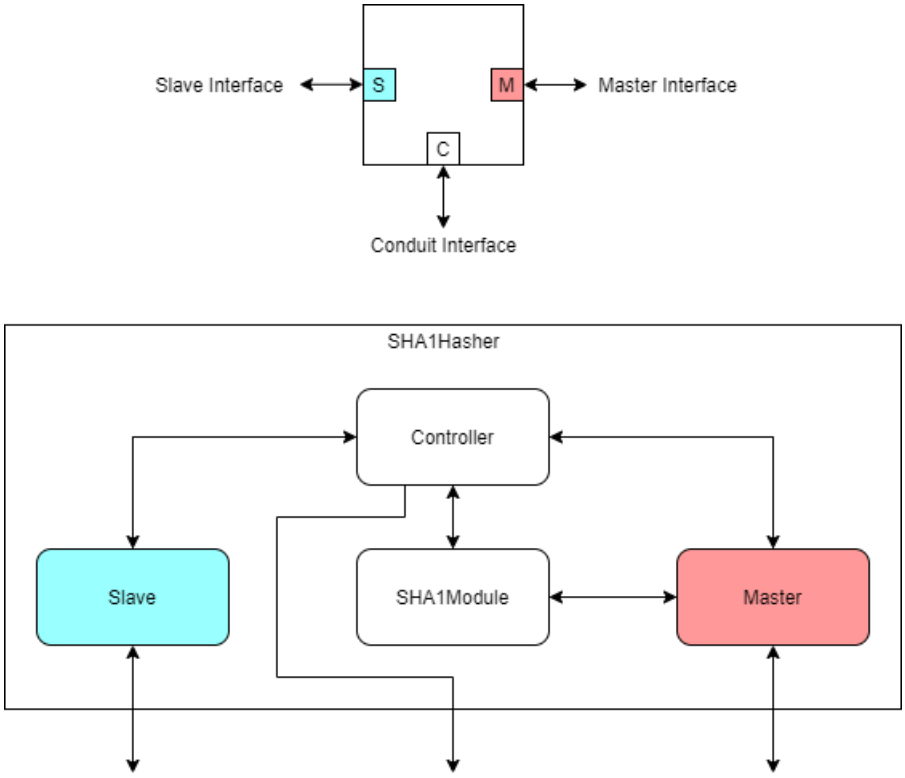

Figure 2: View of the SHA1Hasher's interfaces and submodules.

3.1 Slave

The first sub-module that will be presented is the Slave.
This represents the Avalon Slave interface and it allows the application to program the SHA1Hasher by writing on the registers exposed in the Register Map as well as to read their contents via the NIOS II processor. In particular, the user can specify:

[3]In the following sections of the report we will refer to this IP as Hasher.

- **Read Address**: address in the memory at which the first block is stored.
- **Write Address**: address in the memory at which the first hash is written.
- **Number of Blocks**: number of input blocks.
- **Start**: trigger for the entire accelerator.
- **Complexity**: complexity for the computation of the hashes (i.e. number of '0's that the hashes must have as the most significant bits).
- **Nonce Increment**: integer by which the nonce must be incremented in case an incorrect hash is computed.

Table 1 illustrates the Register Map.

Word Offset	Name	Read/Write	Byte Size
0	read_add_reg	R/W	4
1	write_add_reg	R/W	4
2	num_block	R/W	4
3	start_reg	R/W	4
4	complexity_reg	R/W	4
5	nonce_increment_reg	R/W	4
6	done_reg	R	4

Table 1: Register Map of SHA1Hasher.

3.2 Master

The second subcomponent that will be presented is the Master.
As already said, this module simply implements a DMA to read and write words in the memory.
In particular, since the input blocks are composed of 512 bits and the results of 192 bits (hashes + nonce), the chosen width for the write and read data bus is 64 bits. In such a way, each block can be fetched from the memory with a read burst transfer with burst count equals to 8, and each result can be written in the memory with a write burst transfer with burst count equals to 3. At this point, the implementation of the DMA is straightforward.
When the Controller triggers a read transfer (*start_read* = 1), the DMA sets the *am_read* signal, the *am_address* to the value received by the Controller, and *am_burstcount* to 8. Then, each time the *am_readdatavalid* is asserted, the DMA writes the value on the read data bus in a 512-bit register inside the SHA1Module. Therefore, after 8 64-bit words are read from the memory, the register in the SHA1Module will contain the input block. So, after the read burst transaction has finished, the DMA raises *done_read* and waits for the controller to reset *start_read*. Then, it returns to its initial state.
On the other hand, when the Controller triggers a write transfer (*start_write* = 1), the DMA sets the *am_write* signal, the *am_address* to the value received by the Controller, and *am_burstcount* to 3. Additionally, for each clock cycle in which *am_waitrequest* is 0, the DMA provides on the write data bus the 192-bit result, 64 bits at a time. 160 bits of the 192 are taken from a register in the SHA1Module, as they represent the resulting hash for a block, whereas the other 32 bits are passed by the Controller, as they represent the nonce used to compute that resulting hash. After 3 64-bit words are transferred to the memory, the entire

result has been written. Finally, the DMA raises *done_write* and waits for the Controller to reset *start_write*. Then, it returns to its initial state.
Figure 3 illustrates the states diagram of the FSM for the Master sub-module. Since the state diagram of the FSM is provided we will not present any VHDL snippet, as it would be redundant.

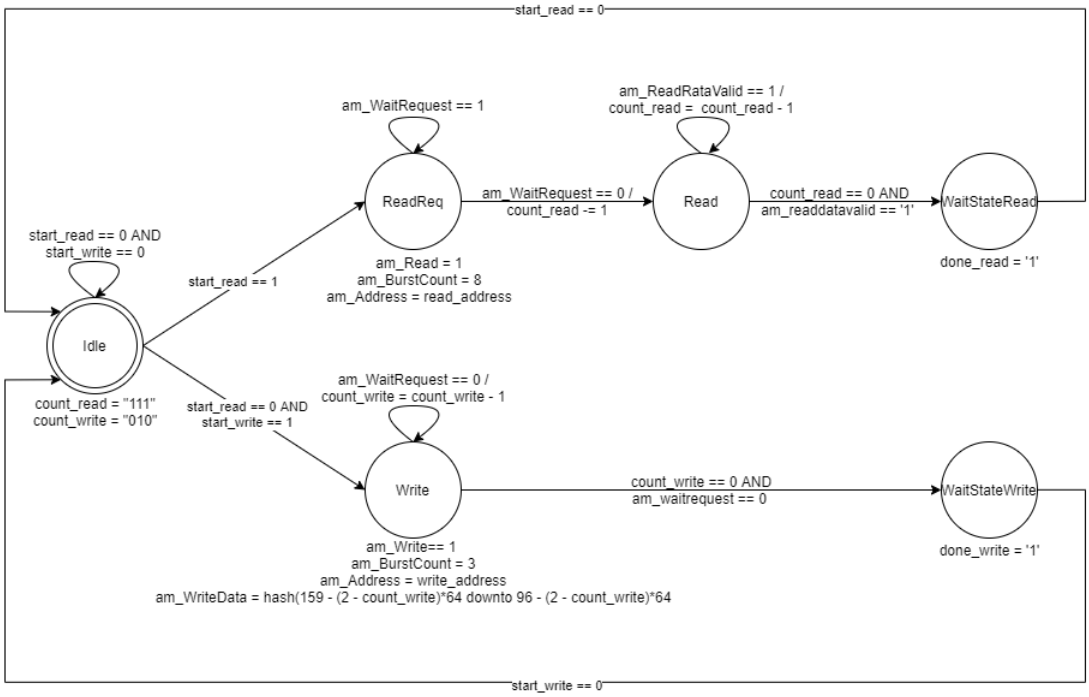

Figure 3: FSM for the Master sub-module.

3.3 SHA1Module

The third sub-module that will be discussed is the SHA1Module.
As previously mentioned, this IP produces the hash for a input block and, as such, it needs to implement the SHA-1 algorithm. For this reason a brief overview of the algorithm will be now given.
The SHA-1 algorithm is mainly composed of two phases, both represented by for loops.
In the first one, an array composed of 80 32-bit words must be populated. Each element of the array is computed using specific portions of the input block. So, this first for loop has exactly 80 iterations.
In the second phase, 5 32-bit words, that represent the internal state, are updated using constants, functions, and the words computed in the first phase. The constants and the functions used for this update vary with respect to the iteration number. Again, this loop has exactly 80 iterations.
Finally, for technical reasons related to the size of the input block, we need to add some extra padding. This means that a new block, composed of fixed patterns, will be used as the input block and then the two main phases of the algorithm will be repeated.
In the end, the hash is simply obtained by concatenating the 5 32-bit words that represent the internal state.
Listing 1 shows the C code for the SHA-1 algorithm.
With this in mind, the implementation of the SHA1Module is straightforward. In particular, we have designed it in VHDL as an FSM.

When the Controller triggers the SHA1Module (*start_hash* = 1), it starts computing the SHA-1 hash using as the input block the content of its 512-bit register, the one on which the Master writes the 64-bit words read from the memory. After the hash has been computed, *done_hash* is asserted and the hash is stored in a 160-bit register that is shared between the Controller and the Master. Finally, the Controller resets the *start_hash* and hence the SHA1Module returns to its initial state.

One final note on the VHDL code. When coding the FSM, we employed variables to mimic the syntax of the provided C code. Although, this may not seem as the typical approach to design hardware, it was very helpful when trying to optimize the algorithm. In particular, instead of computing each iteration of each for loop in one clock cycle we tried to stretch multiple ones in the same clock cycle. After some testing, we were able to compute 4 iterations of the first for loop (the population of the array) in one clock cycle, but unfortunately just one iteration of the second main loop fits in a clock cycle. Nonetheless, this still represents a considerable improvement with respect to the original version. Indeed, if we were to compute each iteration in one clock cycle we would have needed 160 (80 + 80) clock cycles for the first two loops, whereas with the optimization only 100 (20 + 80) are needed. Hence, considering that the two phases of the algorithm are executed twice due to the padding, the SHA1Module needs roughly 200 cycles to compute a hash from an input block. This means that the SHA1Module requires 4 us to compute the hash, since the clock frequency for the entire system is 50 MHz. Since the VHDL code for this module is very similar to the provided C code snippet, no VHDL code nor FSM will be presented.

```c
static void sha1_block(u32* block, struct internal_state* state)
{

    u32 words[80];
    u32 temp;
    //Starts from the previous state
    u32 a = state->A;
    u32 b = state->B;
    u32 c = state->C;
    u32 d = state->D;
    u32 e = state->E;

    //Populates the first 16 words with those extracted from the block and converted in
    ↪  BigEndian
    for(u32 i = 0; i < 16; ++i)
    {
        words[i] = to_big_endian(*(block + i));
    }

    //Populates the remaining words that are calculated from the previous ones
    for(u32 i = 16; i < 80; ++i)
    {
        temp = words[i - 3] ^ words[i - 8] ^ words[i - 14] ^ words[i - 16];
        words[i] = left_rotate(temp, 1);
    }

    //Executes 80 rounds of hashing
    for(u32 i = 0; i < 80; ++i)
    {
        u32 k = round_constants[i / 20];
        u32 w = words[i];
        int f;
        //Depending on the round constants, words and function change
        if (i < 20)
            f = (b & c) | ((~b) & d);
        else if (i < 40)
            f = b ^ c ^ d;
        else if (i < 60)
            f = (b & c) | (b & d) | (c & d);
        else
            f = b ^ c ^ d;

        temp = left_rotate(a, 5) + f + e + w + k;
        e = d;
        d = c;
        c = left_rotate(b, 30);
        b = a;
        a = temp;
    }

    //The internal state is updated with that calculated for this block
    state->A += a;
    state->B += b;
    state->C += c;
    state->D += d;
    state->E += e;

}
```

Listing 1: C code for the SHA-1 algorithm.

3.4 Controller

Finally, the last sub-component that will be dealt is the Controller.
As already mentioned, this component needs to orchestrate the functioning of the others, particularly of the Master and the SHA1Module. Indeed, it is the Controller that triggers and stops these two sub-components. Not only that, but the Controller must also check for the correctness of the hash that the SHA1Module has computed. Again, with correctness we mean the compliance of the produced hash with the complexity specified by the user. In fact, if the computed hash does not respect the complexity, the Controller will change the least significant 32 bits of the input block (**nonce**), and make the SHA1Module start again.
The Controller was implemented in VHDL as an FSM, whose state diagram can be found in figure 4.
As soon as the user writes a 1 in the *start_register*, the Controller asks for a read burst transfer to the Master module, by setting *start_read* and passing the correct read address. When the Master finished the read transfer ($done\_read = 1$), the input block is already stored in the 512-bit register of the SHA1Module. So, the Controller resets *start_read* and makes the SHA1Module start by asserting the *start_hash* signal. When the SHA1Module has computed the hash ($done\_hash = 1$), the Controller needs to check whether it is compliant with the specified complexity. If that is not the case, the Controller will set as the 32 least significant bits of the input block the current value of *count_nonce*, a counter that it is incremented by the value contained in the *nonce_increment_reg* each time an incorrect hash is found. At this point, the Controller will make the SHA1Module start again. When a correct hash is found, the Controller asks for a write burst transfer to the Master, which writes in memory the resulting hash and the nonce used to compute it. This is achieved by setting *start_write* and passing the correct write address. When the write burst transfer is over ($done\_write = 1$), the Controller checks whether there are other blocks to fetch from the memory. If that is the case, the Controller asks for another read burst transfer to the Master and the presented process is repeated. Otherwise, if all the blocks have been correctly hashed, the Controller writes a 1 in the *done_reg* and it asserts its *done* signal, which is exported (Conduit interface). Finally, only when the user resets the value in *start_reg*, the Controller returns to its initial state.
At this point, the reader might be left puzzled by the presented description of the Controller. Indeed, in case no correct hash can be found for a particular block, it seems like the Controller will endlessly try to find a nonce that yields a compliant hash. In order to deal with this corner case, the *count_iter* counter is employed. It simply keeps track of the number of times the Controller has triggered the SHA1Module, and when it reaches 2^{32} it forces the Controller to stop. Finally, to notify the user that the accelerator was not able to find a correct hash, the value "11" is written in the *done_reg* instead of 1.

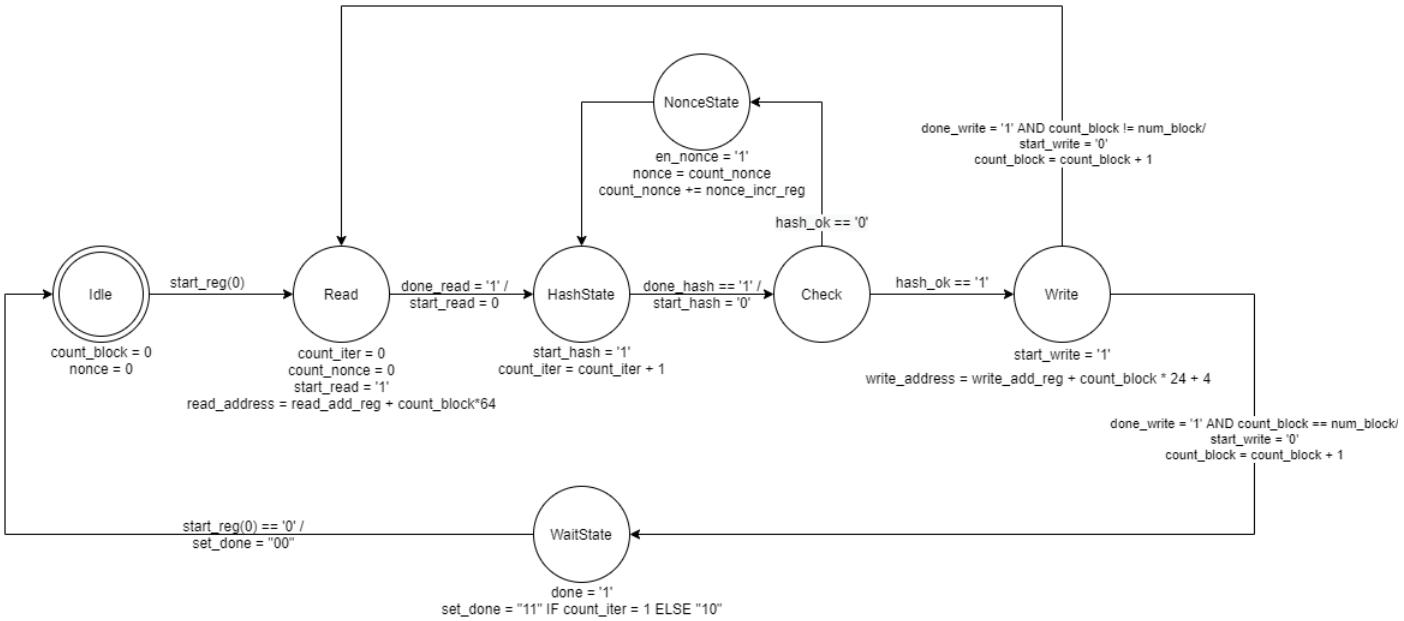

Figure 4: FSM for the Controller sub-module.

4 Custom Instructions

As mentioned in the introduction, one of the goal of the project is to compare the performance between a hardware implementation of the SHA1Hasher, a software one employing hardware custom instructions, and a software-only one. For that reason, 4 custom instructions were coded. They represent some operations in SHA-1 algorithm that are inefficiently executed in software but that can be easily managed in hardware. In our case, these operations are the three rotations and the change of endianness, as can be found in listing 1. Due to their simple VHDL implementation, we will not provide any code snippet.

5 Testbenhches

Before deploying the entire system on the FPGA, some testbenches were carried out to assert the correctness of the presented design of the accelerator. In particular, the different sub components were tested following an incremental approach. That is, we started by testing each of them separately, and then we gradually integrated them in the final version of the Hasher, testing every intermediate step. However, only the final testbench will be presented in this section, as it is the most complete and exhaustive.

The testbench starts by programming the Hasher via the Avalon Slave Interface. That is, it writes the read address, the write address, the complexity, the number of input blocks, and the nonce increment in the corresponding registers in the Slave module. Then, it triggers the Hasher by writing a 1 in its *start_reg*. This can be seen in figure 5.

At this point the Hasher will try to start the read burst transfer, as explained in the previous section. So, the testbench simulates the behaviour of the memory, and provides the 8 64-bit words composing the first input block on the read data bus while asserting *am_readdatavalid*. Moreover, the *am_readdatavalid* signal is reset for some clock cycles, to verify whether the Master sub-module behaves correctly. This is illustrated in figure 6.

Then, the Hasher computes the correct hash (compliant with the specified complexity), and when it has finished it triggers a write burst transfer. As can be seen in figure 7, the

am_burstcount and *am_address* signals are set to their correct values for the entire transaction. The resulting hash and the employed nonce are provided on the write data bus as 3 64-words, while asserting the *am_write* signal.
Since the number of blocks specified by the testbench was 2, the Hasher now starts another read burst transfer. Again, the testbench simulates the behaviour of the memory and provides the 8 64-bit words, similarly to how described before. The only difference is in the 8 input words themselves. This is shown in figure 8.
Now, the Hasher will compute the correct hash for the received block, and it will then execute the last write burst transaction. This is performed in the same way as the first one. Finally, since there are no block left to hash, the Hasher will rise its *done* signal, which is reset only when the testbench writes a zero value on the *start_reg*. All of this can be seen in figure 9.

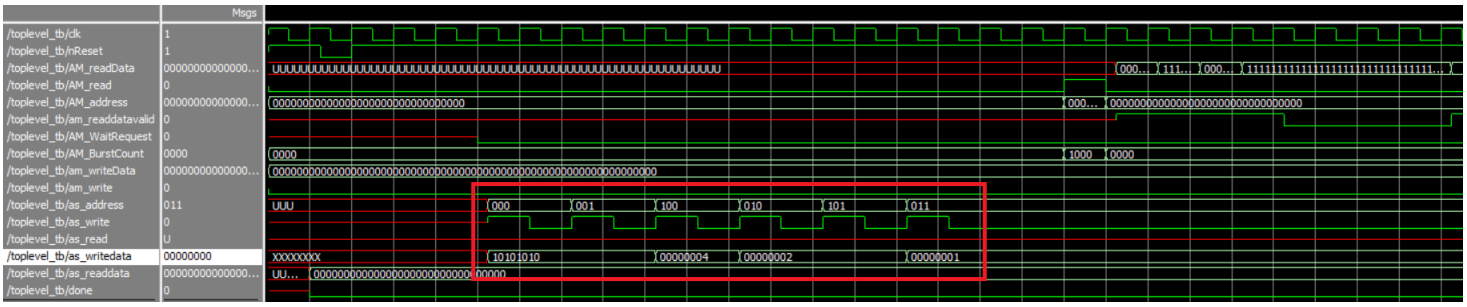

Figure 5: Programming the Hasher.

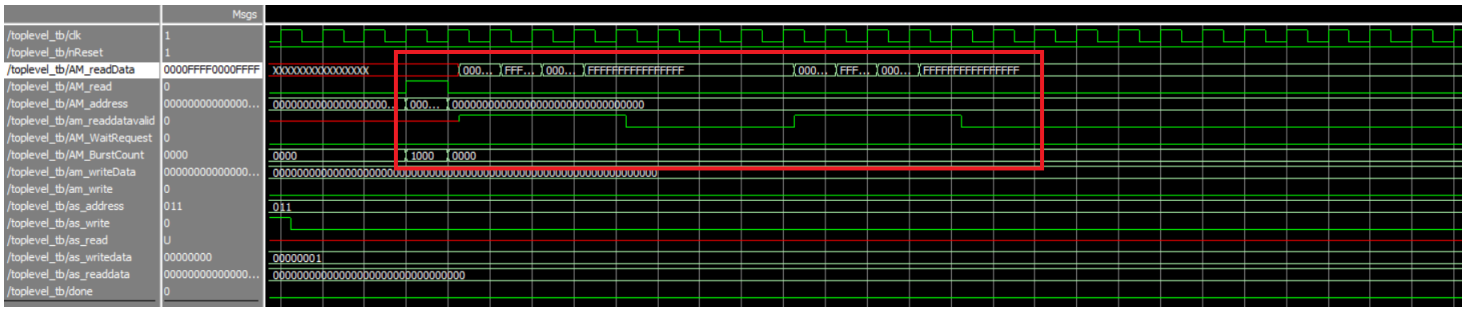

Figure 6: First read transfer from the memory to the Hasher.

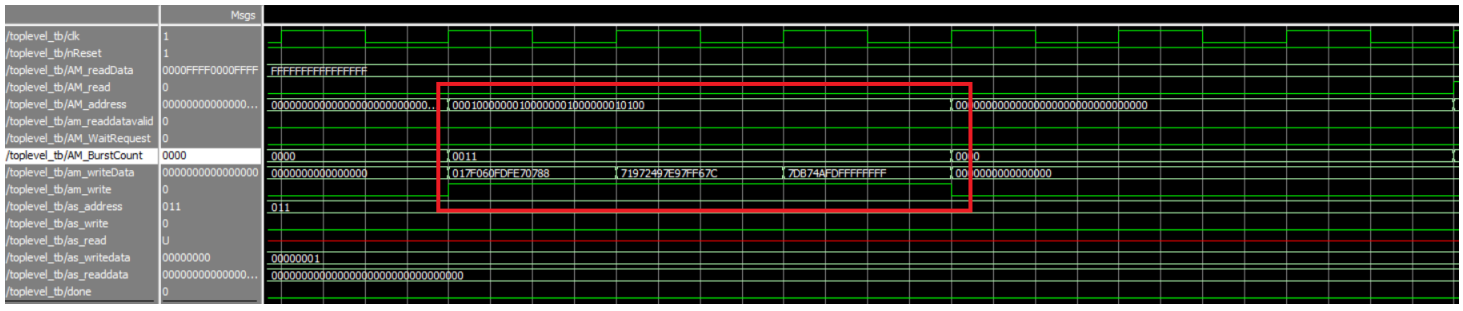

Figure 7: First write transfer from the Hasher to the memory.

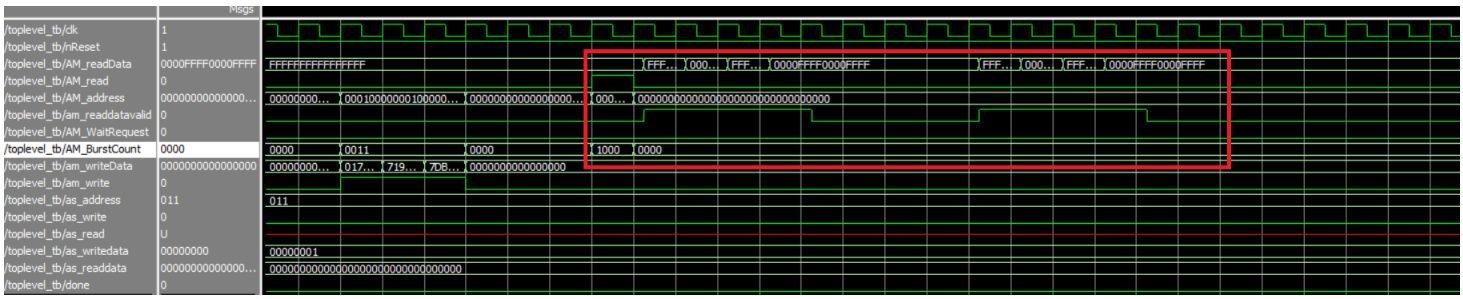

Figure 8: Second read transfer from the memory to the Hasher.

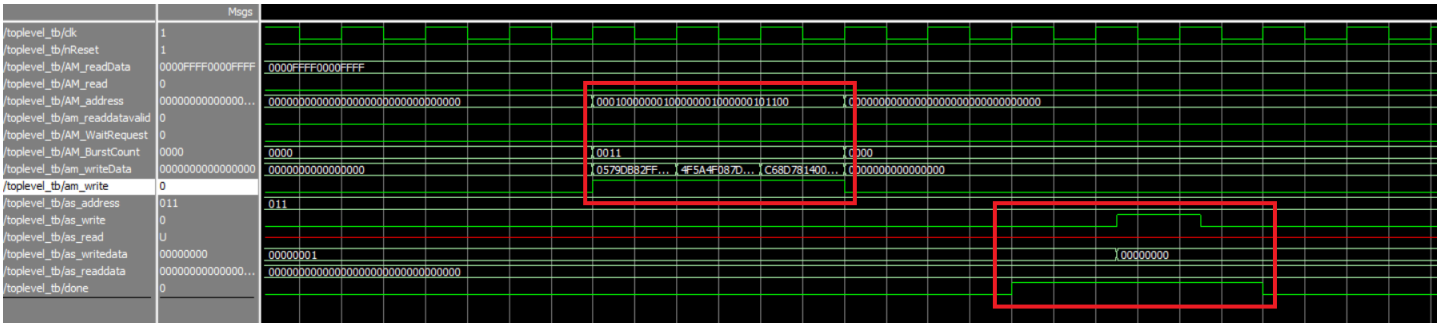

Figure 9: Second write transfer and setting of the *done* signal.

6 Deployment on the board

In this section, a quick overview of the mapping of the conduit signals of the system with the peripherals of the DE1-Nano-SoC will be provided.

- PIO(3 downto 0) : GPIO_0(7 downto 4).
- PIO(7 downto 4) : LEDR(3 downto 0).
- *done* signal Hasher_0 : GPIO_0(0).
- *done* signal Hasher_1 : GPIO_0(1).
- *done* signal Hasher_2 : GPIO_0(2).
- *done* signal Hasher_3 : GPIO_0(3).
- *nReset* signal : KEY_N(0).

7 C code

In this section, the C code will be analyzed. As mentioned in the introduction section, the goal of the project is double. On the one hand, we aimed at simulating the miners of a block chain employing a simplified version of the proof-of-work technique. On the other hand, we profiled the performance of a hardware solution, a software with hardware custom instructions solution, and a pure software solution to measure the difference in time that each approach requires. For this reason, two C files were produced.

7.1 Demo.c

The first C code that will be analyzed is Demo.c. This is the one implementing the simulation of the miners of a block chain employing a simplified version of the proof-of-work technique. The code is structured in such a way that the user can easily change parameters as the number of input blocks, the number of working hashers (maximum 4), the complexity, and the nonce increment for each hasher. Indeed, these parameters are expressed as global variables.
Firstly, the PIO is reset and its direction is set to output. Them, some input blocks are written in the memory, starting from its initial address. This is taken care of by the *write_blocks()* function.
At this point the main loop of the code begins.
For starters, every Hasher is programmed. That is, the application provides the values for the initial read address, the initial write address, the number of blocks, the complexity and the nonce increment. This is implemented in the *set_hasher()* function. Then, each Hasher is triggered (write a 1 in the *start_reg*), and different pins of the PIO are set. This is used only to probe the *done* signal of each Hasher via the logic analyzer, but it is not strictly necessary for the functioning of the code.
It is worth noting that although the Hashers are started sequentially (from 0 to 3), this does not introduce any bias in the game. Indeed, one might think that since the fourth Hasher is started for last, it has a clear disadvantage over the others. Nonetheless, some measurements were taken to quantize the time that elapses from the start of the first Hasher to the start of the fourth one. The measured amount of time is around 1 us, which is less than the computation of one hash for one block (as discussed in section 3.3).
Now, that all Hashers are started, the application polls on all of the done registers. As soon as one Hasher has finished ($done\_reg = 1$), the application compares the hashes computed by that accelerator with the one produced by the sha.h file. If all the hashes are correct, the application will reward the Hasher by assigning it one point. This is taken care of by the *read_from_memory* function, whose snippet can be seen in listing 2. Note that the Hashers write 64-bit words in memory and that the hash function coded in sha.h requires the input blocks to be expressed in Big Endian. This should explain the unusual offset for reading from the memory.
At this point, the application needs to wait for the other Hashers to finish and then it increments the complexity. This determine the end of one round. Then, the entire process is repeated until the maximum complexity (another global variable) is reached. In the end, the Hasher that has gained more points is the winner of the game and its corresponding led will be switched on.
Due to the length of the main loop, we prefer not to provide any snippet of it.

```
uint32_t offset = num_blocks*64 + num_hasher*num_blocks*24 + ind_curr_block*24;

hash[0] = IORD_32DIRECT(ONCHIP_MEMORY2_0_BASE + offset, 4);

hash[1] = IORD_32DIRECT(ONCHIP_MEMORY2_0_BASE + offset, 0);

hash[2] = IORD_32DIRECT(ONCHIP_MEMORY2_0_BASE + offset, 12);

hash[3] = IORD_32DIRECT(ONCHIP_MEMORY2_0_BASE + offset, 8);

hash[4] = IORD_32DIRECT(ONCHIP_MEMORY2_0_BASE + offset, 20);

hash[5] = IORD_32DIRECT(ONCHIP_MEMORY2_0_BASE + offset, 16);


//check with software
uint8_t input_hash[64];
uint8_t ind;
for(ind = 0; ind < 60; ind+=8){
        input_hash[ind] = IORD_8DIRECT(ONCHIP_MEMORY2_0_BASE, ind_curr_block*64 + ind
        ↪  + 7);
        input_hash[ind + 1] = IORD_8DIRECT(ONCHIP_MEMORY2_0_BASE, ind_curr_block*64 +
        ↪  ind + 6);
        input_hash[ind + 2] = IORD_8DIRECT(ONCHIP_MEMORY2_0_BASE, ind_curr_block*64 +
        ↪  ind + 5);
        input_hash[ind + 3] = IORD_8DIRECT(ONCHIP_MEMORY2_0_BASE, ind_curr_block*64 +
        ↪  ind + 4);
        input_hash[ind + 4] = IORD_8DIRECT(ONCHIP_MEMORY2_0_BASE, ind_curr_block*64 +
        ↪  ind + 3);
        input_hash[ind + 5] = IORD_8DIRECT(ONCHIP_MEMORY2_0_BASE, ind_curr_block*64 +
        ↪  ind + 2);
        input_hash[ind + 6] = IORD_8DIRECT(ONCHIP_MEMORY2_0_BASE, ind_curr_block*64 +
        ↪  ind + 1);
        input_hash[ind + 7] = IORD_8DIRECT(ONCHIP_MEMORY2_0_BASE, ind_curr_block*64 +
        ↪  ind);

}
input_hash[60] = IORD_8DIRECT(ONCHIP_MEMORY2_0_BASE + offset, 19);
input_hash[61] = IORD_8DIRECT(ONCHIP_MEMORY2_0_BASE + offset, 18);
input_hash[62] = IORD_8DIRECT(ONCHIP_MEMORY2_0_BASE + offset, 17);
input_hash[63] = IORD_8DIRECT(ONCHIP_MEMORY2_0_BASE + offset, 16);


struct internal_state res_sha = sha1(input_hash, 64, 0);
```

Listing 2: read_from_memory function.

7.2 Timing.c

The second C code that will be dealt with is Timing.c. This is the code used to profile the performances of the Hasher accelerator, the software with custom instructions, and the software without custom instructions. Again, the code was designed in such a way that the user can carry out different experiments by simply changing some parameters expressed as global variables. These are: the number of blocks, the complexity, and the nonce increment. The rationale behind the code is that the three different implementations are asked to compute correct hashes (i.e. compliant with the specified complexity) for increasing complexity and increasing number of blocks. Obviously, this is achieved with nested loops.

In particular, in every iteration of the external loop the following happens:

- One block is written in the memory after the ones written in the previous iterations.
- The Hasher is programmed and triggered to compute correct hashes for the input blocks for increasing complexity. For every complexity experiment, the first pin of the PIO is set when the Hasher starts and it is reset when it has finished.
- The software with custom instructions computes hashes for the input blocks for increasing complexity and also needs to check for their correctness. For every complexity experiment, the second pin of the PIO is set when the software starts and it is reset when it has finished.
- The software without custom instructions computes hashes for the input blocks for increasing complexity and also needs to check for their correctness. For every complexity experiment, the third pin of the PIO is set when the software starts and it is reset when it has finished.

This is repeated until the maximum number of blocks is reached. Again due to the length of the code, we prefer not to provide any snippet of code. However, the result of the execution as seen on the logic analyzer can be found in figure 10. The first channel corresponds to the done signal of the Hasher, the second one to the first pin of the PIO, the third one to the second pin of the PIO, and the fourth one to the third pin of the PIO.

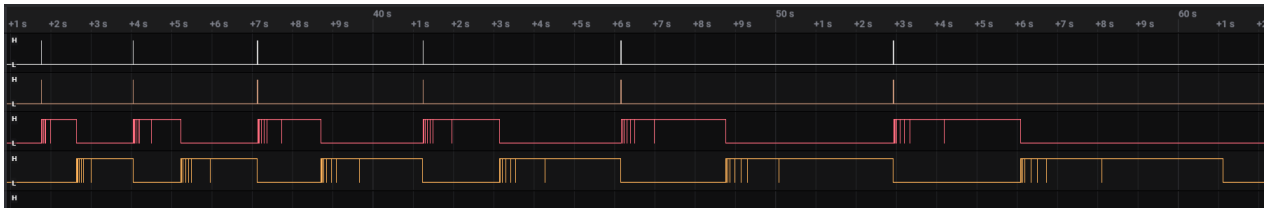

Figure 10: Result of the execution of Timing.c as seen on the logic analyzer software tool.

8 Measurements

In this section of the report, we will analyze the measurements taken by running Timing.c and probing the pins of the GPIO. We will firstly compare the difference in performance of the three implementations, and then we will try to investigate the mathematical relations between the time required to compute the hashes, the number of blocks and the complexity. For starters, the performance of the three different implementations are summed up in figures 11, 12, and 13. As can be clearly seen, the fully hardware solution is much faster than the software implementations. Indeed, in the worst case (complexity = 8 and number of blocks = 6) the time required by the Hasher is roughly 250 times less than the one required by the software using custom instructions and 400 times less than the one required by the software without the custom instruction. Additionally, using hardware custom instructions for the software-heavy operations of rotations and change of endianness seemed to yield better performance. Nonetheless, the gain is not as high as one might expect it to be. Indeed, in almost all the different configurations the time required by the software employing the custom instructions is only 1.5 times less then the one required by the software-only implementation. This suggests that there might be other instructions that the NIOS II struggles to execute efficiently. Some candidates might be the bit-wise operations in the main loop of the sha algorithm, as presented in section 3.3. Being aware of that, we tried to produce custom

instructions for these operations, as well. Nonetheless, it is only possible to create custom instructions with maximum 2 32-bit input words. Using such custom instructions to re-write the bit-wise operations of the main loop unsurprisingly resulted in worse performance. Hence, only the custom instructions for the rotation and change of endianness were employed. Figure 14 provides an interesting summary of what just presented.

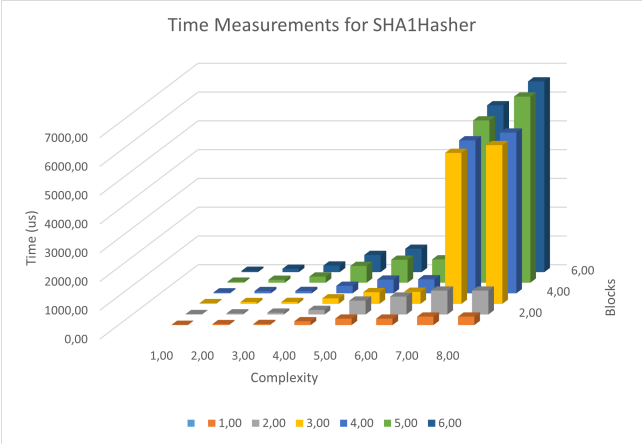

Figure 11: Performance for Hasher.

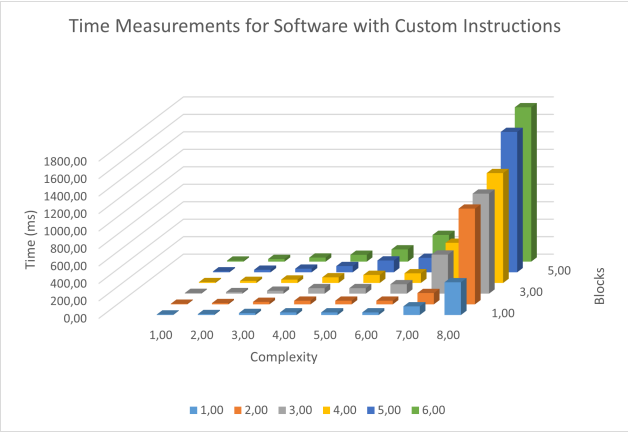

Figure 12: Performance for software with custom instructions.

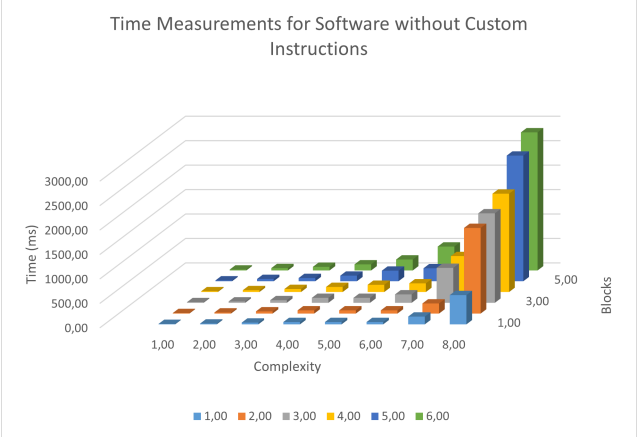

Figure 13: Performance for software without custom instructions.

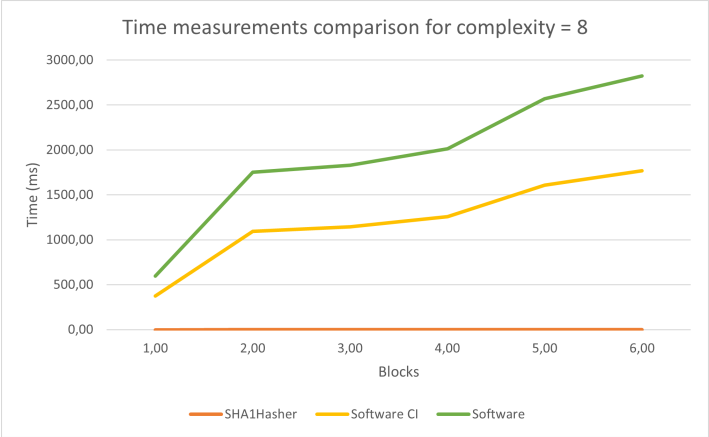

Figure 14: Comparison of performance for the three implementations for complexity = 8.

At this point, we will deal with the relations that exist between the performance, the number of blocks and the the complexity.
Firstly, the relation between the required time and the number of blocks seems to be linear, as the time to hash a single block is fixed. This is supported by the collected data. As an example, a chart representing such linear relation for the software with custom instructions in case of complexity equals to 5 is shown in figure 15. Nonetheless, one should not forget that the measured time is not the one for the computation of mere hashes, rather it is the one for the computation of hashes compliant with the specified complexity. Hence, it can happen that by simply adding one new block, the required time grows more than linearly, as the new correct hash might be difficult to find. This is especially true for high complexities, and the data collected for all of three implementations support this claim.
Secondly, the relation between the required time and the complexity appears to be exponential. This seems reasonable as by increasing the complexity by 1, the number of compliant hashes is halved. Again, this claim finds support in the collected data, and the software with custom instructions is again taken as reference for the plot in figure 16. However, it is worth noting that there could be cases in which an increase in complexity does not yield any increase in terms of required time. The reason for that lies again in the functioning of the proof-of-work technique. Indeed, a hash having x zeros as its most significant bits will be compliant with all complexity from 1 to x. So in case of only one input block, if such a hash is found when the

complexity is $y, y < x$, the required time will not increase from round y to round x. Again, such behavior is reflected in the collected data.

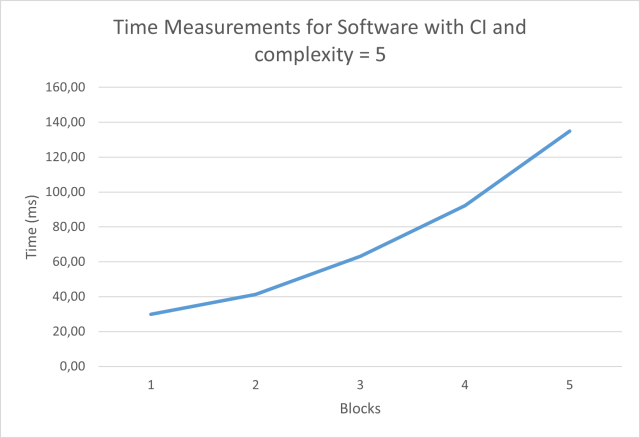

Figure 15: Evolution of the required time over the number of blocks for the software with custom instructions and complexity = 5.

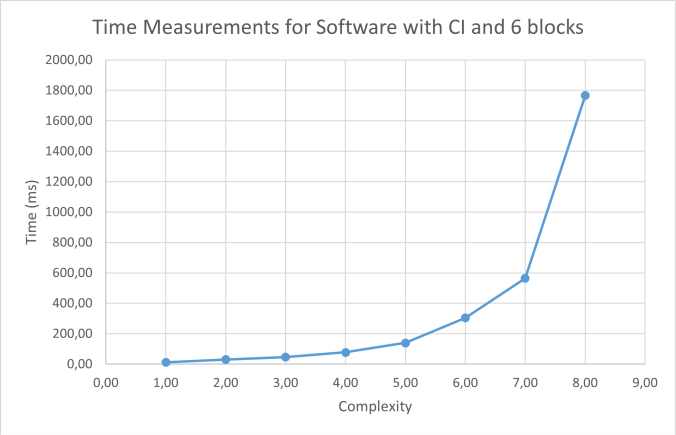

Figure 16: Evolution of the required time over the complexity for the software with custom instructions and number of blocks = 6.

9 Conclusions

In this report an in-depth analysis of the entire project was presented. Not only were we able to simulate correctly the miners of the block chain in a game-like manner, but we also managed to extract data and draw some conclusions on the difference in performance between the presented implementations. One possible improvement might be the collection of a higher number of data so to strengthen our results.