EE6470 Electronic System Level Design and Synthesis

Final Project Report

108062209王鴻昱

* Github repo: <https://github.com/PaulWang0513/Electronic-System-Level-Design-and-Synthesis>
* Problem Description and Solution

The requirement of final project is divided into two parts. For the first part, we need to choose an algorithm and implement the HLS accelerator PE, as well as the testbench, then optimize the PE and compare the designs. For the second part, we need to port the implemented PE onto the riscv-vp platform, implement the software on the multi-core processors, and compare some DMA information.

The algorithm I choose is autocorrelation function, a tool that is commonly used to analysis the periodic information of time-serial signal. I first implemented a basic HLS accelerator, then try to optimize in several ways for the first part. For the second part, I modified the codes to the tlm version to port onto the riscv-vp platform.

* Implementation Details
* Autocorrelation Function (ACF)

The ACF is a mathematical tool that measures the similarity of a signal to itself with different level of delay. Specifically, for a signal with N data points, the ACF performs N output results. The first result is the correlation of the original signal and the signal with no delay, i.e. the same signal. The second result is the correlation of the original signal and the signal with 1 delay, so and so on.

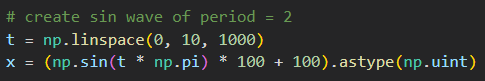
The formula can be written as:

where indicates the delayed value, indicates the value of signal at time , and indicates the length of signal .

ACF is commonly used to estimate the periodic information of a signal. Intuitively, if a signal have period , then should be a large value.

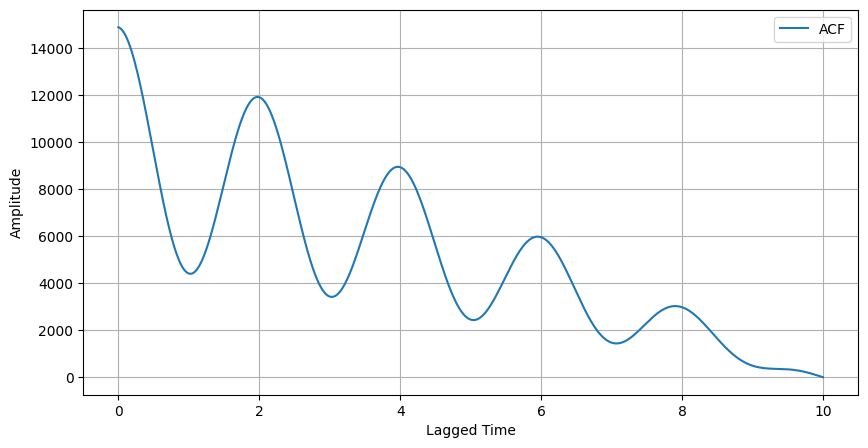
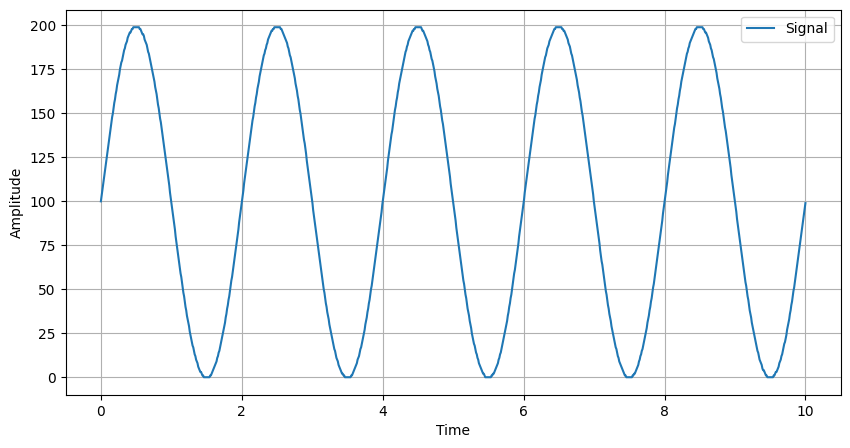
* Test data

The data I used to verify the implementation is a sin wave with period of 2 seconds. The signal is 10 seconds long with 100 data points each second. The signal value is quantized as unsigned integer in the range 0 to 200, so they can be represented in 8 bits.



*Image 1. Code to generate the test data*

*Image 2. Test data (left) and the ACF result (right)*



* Specification

Input: 8-bit unsigned integer.

Output: 26-bit unsigned integer.

The number of bits is designed to avoid overflow problem. The largest possible value is 1000\*255\*255 = 65025000 < 226, where 255 is the largest value of an 8-bit unsigned integer.

Memory usage: at least 1000\*8-bit to store the signal.

* Basic Implementation

In the basic implementation, my goal is to designed a functional accelerator with simple architecture. I used a single-port memory with WordSize=8, NumWords=1000, and Area=1000, which is named as RAM\_1000X8.

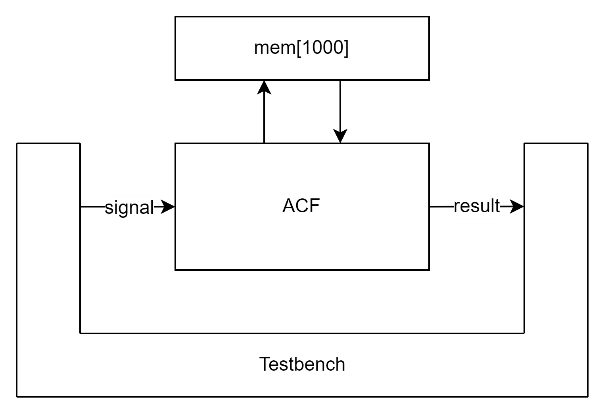
Some other single-port memory block, such as RAM\_500X8 and RAM\_250X8, will be used in latter implementations. The naming shows the information of the memory (RAM\_[NumWords]X[WordSize]), and the area is set as same with NumWords.

The dataflow of this ACF PE can be separated into the following 3 parts:

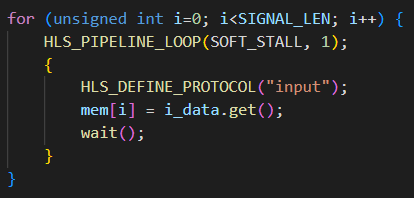
(1) Load 1000 signal data from testbench, then store into the memory.

(2) To compute the result i, read data from memory and sum up.

(3) Output the averaged result i. Go back to (2) for result i+1 if i<N-1.

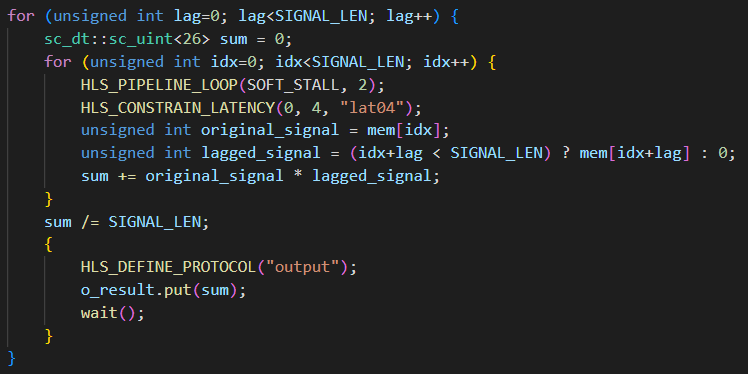


*Image 3. System architecture of the basic implementation*

For loading 1000 signals and store into memory, it takes at least 1000 cycles to complete because we can only get 1 data from testbench and store 1 data into memory in each cycle. Image 4 shows the actual implementation. Although the loop bound is a constant, the constraint of data access make loop unrolling unachievable.

*Image 4. Code of data reading and storing in basic implementation*

After all data is ready in memory, we can start computing each ACF result of the input signal. This can be easily implemented within two loops in image 5. In each iteration of the outer loop, it computes the result with certain lag value and output to testbench eventually. The inner loop accumulate the point-to-point multiplication of the original singal and lagged signal. Because the original\_signal and the lagged\_signal both access data from mem, at least 2 cycles are needed in each iteration, and the initial interval is set to 2 because of this. The usage of HLS\_PIPELINE\_LOOP directive and HLS\_CONSTRAIN\_LATENCY directive ensure the inner loop to finish in 2002 cycles (2\*SIGNAL\_LEN + 4-2).

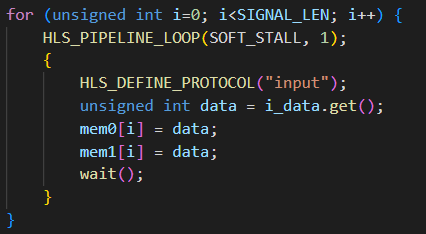


*Image 5. Code of computing results in basic implementation*

In summary, the basic implementation needs at least 2000 cycles to compute an ACF result, which is limited by the memory access.

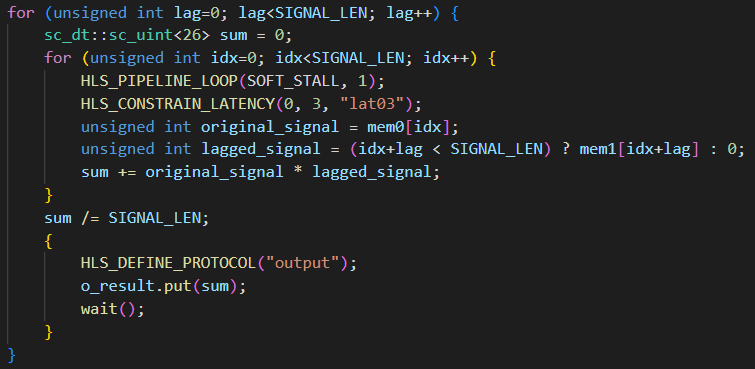
* MEM\_1000X2 implementation

In order to reduce the total latency of memory access, the parallelism need to be improved. The most intuitive way is to use another memory to store the duplicated data. In this implementation, I use two RAM\_1000X8 to store copy of the signal, so two data can be read from the memories in one cycle.



*Image 6. Code of data reading and storing in MEM\_1000X2 implementation*

Same with the basic implementation, reading data from testbench and store into memories still need at least 1000 cycles. But in the computation step, we only need one cycle for memory access in the inner loop, so the initial interval can be set to 1.



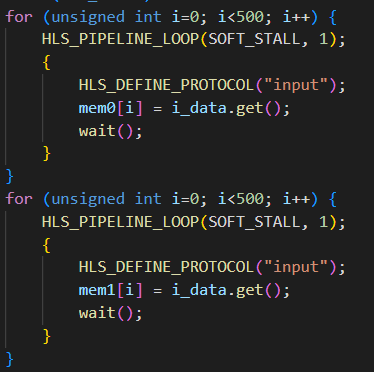
*Image 7. Code of computing results in MEM\_1000X2 implementation*

In summary, the MEM\_1000X2 implementation needs at least 1000 cycles to compute an ACF result, with additional area for the second memory.

* MEM\_500X2

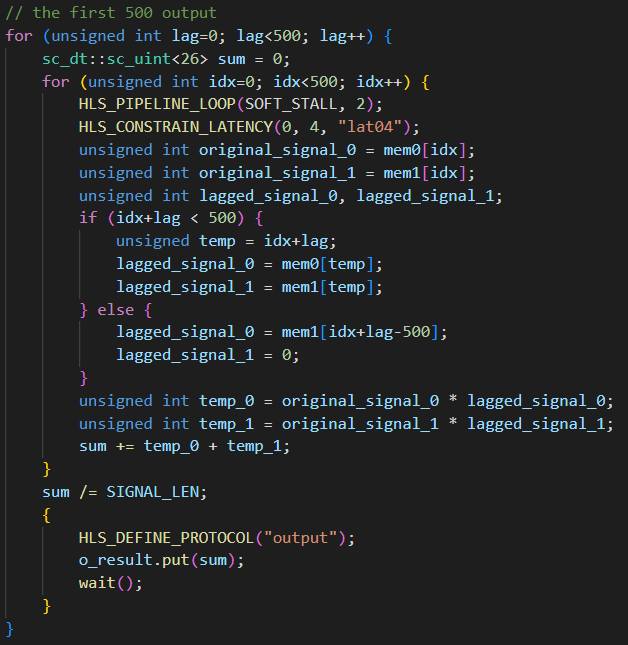
It seems a bit wasting to use two memories to store the same data, so in this implementation, I use two RAM\_500X8 to store the 1000 input data.

By storing the first 500 data in mem0 and the last 500 data in mem1, two memory access can be done in one cycle. However, different with MEM\_1000X2 implementation, we cannot get the desired data every time since one should come from the first memory, and the other should come from the second memory.



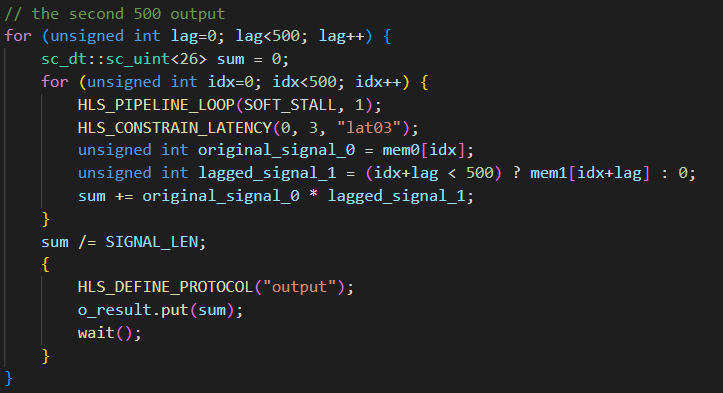
*Image 8. Code of data reading and storing in MEM\_500X2 implementation*

Because of this limitation, we need to modify the algorithm to take advantage of it. For the first 500 ACF result, the two data to be multiplied and accumulated are in the same memory. We use 1 cycle to read two original\_signal data from mem0 and mem1, and use another cycle to read the two-corresponding lagged\_signal data from mem0 and mem1. With initial interval set as 2, we can still use 1000 cycles obtain the data for computing an ACF result.



*Image 9. Code of computing the first 500 results in MEM\_500X2 implementation*

And for the last 500 ACF result, the two data to be multiplied and accumulated are located in different memory. Furthermore, we only need to accumulate half of the multiplied result since the other 500 will eventually become 0. With proper pipeline, only 500 cycles are needed to read data for an ACF result in the last 500 ones.

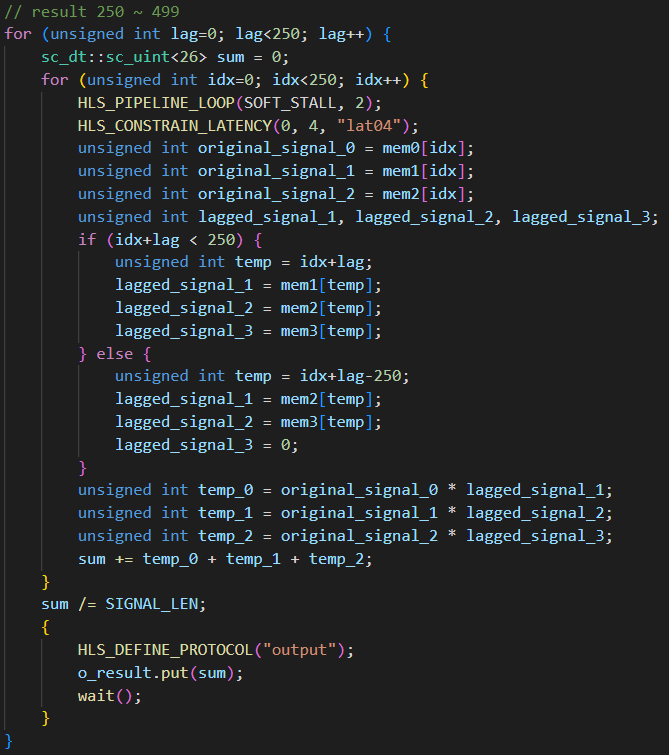


*Image 10. Code of computing the last 500 results in MEM\_500X2 implementation*

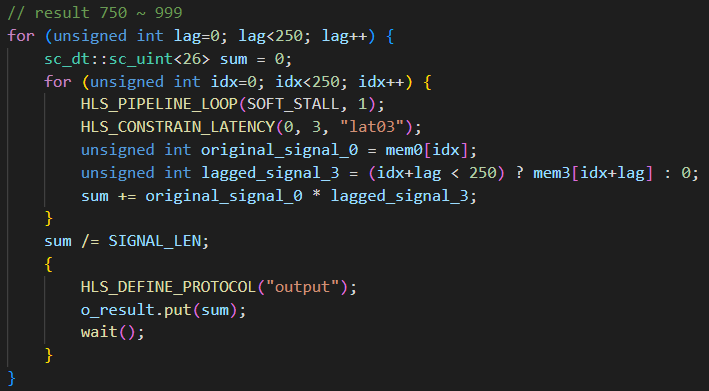
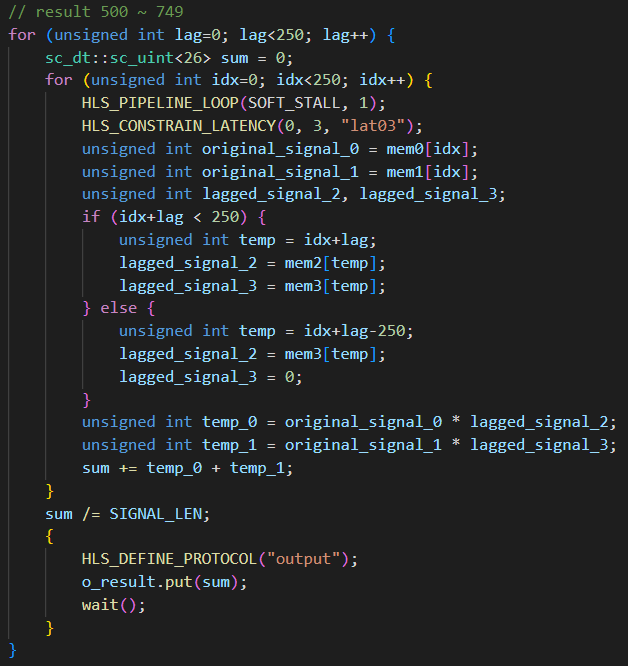
In summary, by properly arranging the memory usage, the number of cycles for memory access decrease to even less than MEM\_1000X2 implementation (750 cycles in average).

* MEM\_250X4

With the same thought, we can further divide the memory into 4, and use 4 outer for-loop to compute 4\*250 ACF result. By this, we need only 500 cycles for memory access when computing the first 500 result, and only 250 cycles for memory access for the last 500 result.



*Image 11. Code of computing the first 500 results in MEM\_250X4 implementation*

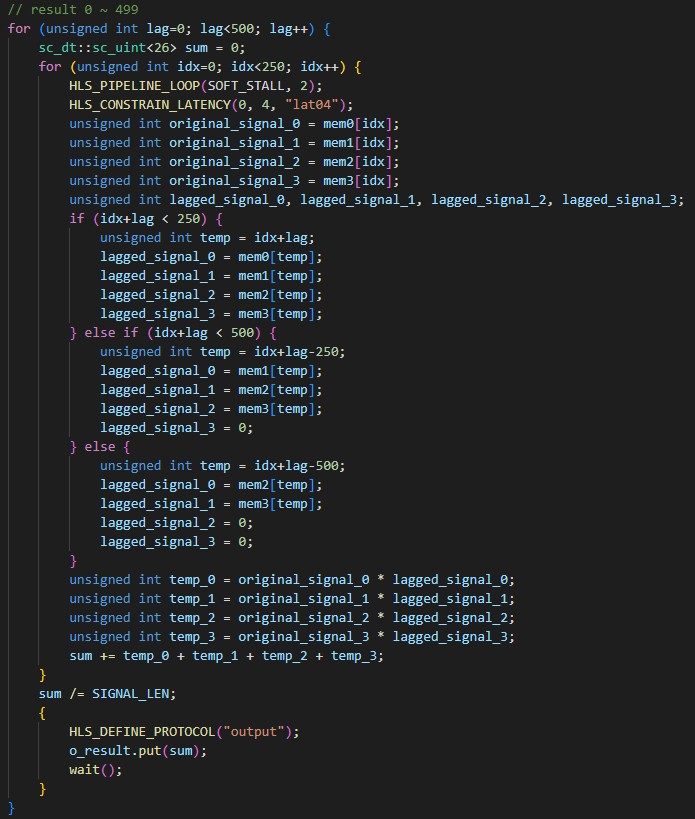


*Image 12. Code of computing the last 500 results in MEM\_250X4 implementation*

* MEM\_250X4\_AREA

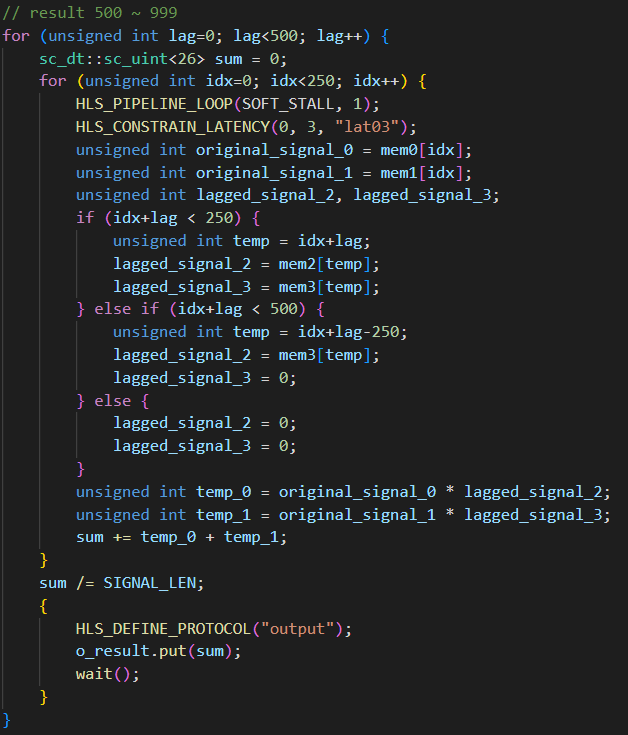
In MEM\_250X4 implementation, although we achieved a relative low latency, the area grew badly because the more complex implementation way of the algorithm.

To reduce the complexity, we can combine the first two loops together, and the two last loops together since they take the same latency in memory access.



*Image 13. Code of computing the first 500 results in MEM\_250X4\_AREA implementation*

After this modification, the latency of computing an ACF result increase a little bit, because of the more complex logic of the inner loop, which aim to decide the memory to access. Nevertheless, the area of the design does decrease as we thought.



*Image 14. Code of computing the last 500 results in MEM\_250X4\_AREA implementation*

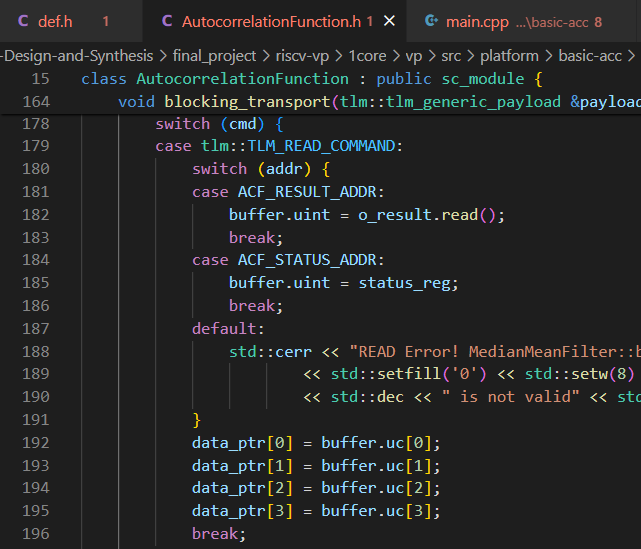
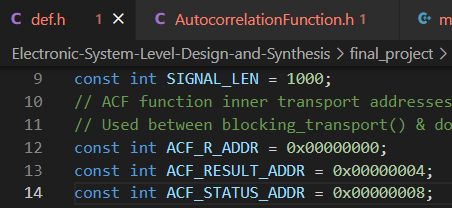
* risv-vp – single core

To port the PE on the risv-vp platform, I need the modify the implementation with TLM coding style. I used the MEM\_250X4\_AREA implementation to do the modification.

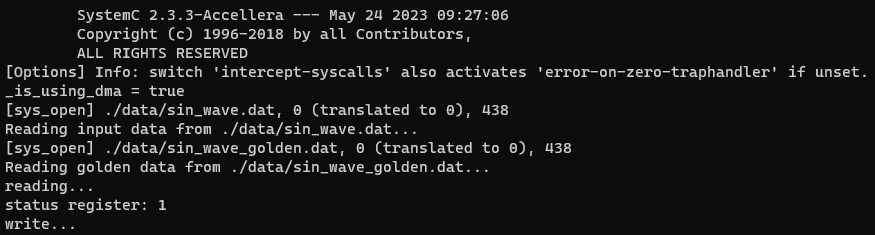
There is no much difference in coding. First, I changed the interface of the module from cynw\_p2p to sc\_fifo, created target socket of the module, specify the reading and writing address, remove the unsupported HLS directives, add the blocking transport function which is almost the same with one in hw4, and insert some wait to make the simulation go smoothly. These steps are not special since we have done in homework. Then I modified the main.cpp for vp platform, it is basically replacing the MedianMeanFilter module with the ACF module.

So far, the platform code is finished and can easily control by the software testbench with DMA. In each DMA transaction, I move 4 bytes at a time (when writing data, only 8 bits are meaningful; when reading result, only 26 bits are meaningful) with 4-byte DMA bandwidth. Assume the initial cycle is 2, the total DMA cycle

Next, I tried to add a status register into the module to compute the simulated time, but encounter some problem. As other interface, I add an address for the register, declare the variable, and add the blocking transport function case for the status register. The register does become accessible from software by DMA now.

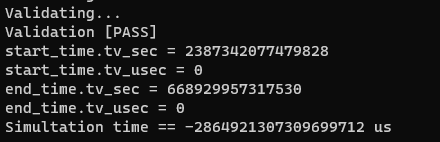


*Image 15. Part of code for the status register implementation*



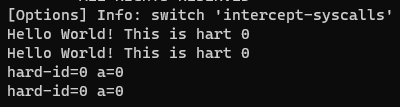
*Image 16. Message in the terminal from software that shows the value of status register*

Then I tried to record the simulated time with sc\_timestamp(), but find out the systemC library cannot be find from software path. So, I turned to tried using struct timeval to at least count the simulation time. However, it gave me an unreasonable simulation time that I can’t find out the reason.



*Image 17. Unreasonable time recording. The actual simulation time should be within 10 seconds.*

* risv-vp – multicore

In this part, I tried to implement the multicore platform but failed eventually. To build a multicore platform, I imitated the tiny32-mc to modify the code of the single core platform. Basically, I just double the things about core, ACF, and DMA. After the modification, I used the software code basic-multicore-printf to test the platform code, but find that it doesn’t work as expected QQ.

*Image 18. Test result of the multicore platform with basic-multicore-printf software*

* Additional Features

The HLS code is written with compiler directive (#if defined, #elif defined, and #endif), so different implementation can be agilely switch and test within 2 files (\*.h and \*.cpp file).

* Experiment Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Implementation | Clock period (ns) | Area | Avg. latency (cycle) | Total runtime (ns) |
| BASIC | 10 | 3843.2 | 2002 | 20040010 |
| MEM\_1000X2 | 10 | 4749.2 | 1002 | 10040010 |
| MEM\_500X2 | 10 | 5675.3 | 752 | 7540030 |
| MEM\_250X4 | 10 | 9969.6 | 377 | 3790070 |
| MEM\_250X4\_AREA | 10 | 7999.7  *Table 1. comparison between different implementation* | 380 | 3812570 |

Table 1 show the comparison of area, average latency, and total runtime between different implementation. The average latency comes from averaging the number of cycles between each ACF result output.

In the BASIC implementation, it needs at least 2000 cycles of memory access to compute one ACF result, the average latency of 2002 cycles prove this thought. The runtime comes from (2002\*1000 + 2001) \* 10 (ns), where the 2001 cycles include 1000 cycles of reading data from testbench, and 1000 cycles of output result to testbench.

In the MEM\_1000X2 implementation, the average latency decreases to almost half of the one in BASIC implementation, with the area larger by almost 1000 (the size of RAM\_1000X8).

In the MEM\_500X2 implementation, the average latency further decreases to 752 cycles, almost (1000+500)/2, and has a larger area because of the more complex control flow than one in MEM\_1000X2 implementation.

In the MEM\_250X4 implementation, with the same thought of MEM\_500X2 implementation, it achieves an even smaller average latency and larger area.

In the MEM\_250X4\_AREA, we simplified the control logic to make the area become smaller, while only increase the average latency for a little bit.

* Discussion and Conclusions

In this final project, I proposed and analysis a way to tradeoff between area and speed in the ACF PE implementation. This thought may also be applied on other PE that rely on much of memory access with simple computation.

At the second part, I read much of the source code of the riscv-vp to implement the multicore platform. Although I didn’t complete a functional one, the research did give me the opportunity to learn much of unknown knowledge.