EE6470 Electronic System Level Design and Synthesis
Homework2 Report

108062209 王鴻昱

- Github repo: https://github.com/PaulWang0513/Electronic-System-Level-Design-and-Synthesis

- Problem Description and Solution

  Part0:

  Originally, I implemented the median filter and mean filter as two methods (processes) in a same class (module), both read data from the same FIFO channels and use two enable signals to control the processing order. However, I discovered that this implementation may cause problem in later steps, so I modified the communication way to eliminate two enable signal, by separate the used channels.

  Part1:

  In this part, we are asked to wrap the filter module with TLM interface, and insert wait() in the target module to model the delay. I reused the Initiator.h and Initialtor.cpp in lab3, implemented my own blocking transport function in target module, and modified other files to match the TLM coding rules.

  Part2:

  In this part, we are asked to setup a quantum keeper base on the previous implementation to reduce the wait() in simulation. So, I modified the Initiator module to allow the transport of delay, added a quantum keeper in the testbench, and changed the wait() in target module to delay. Also, I have tried different value of quantum to see the difference of simulation time.

  Part3:

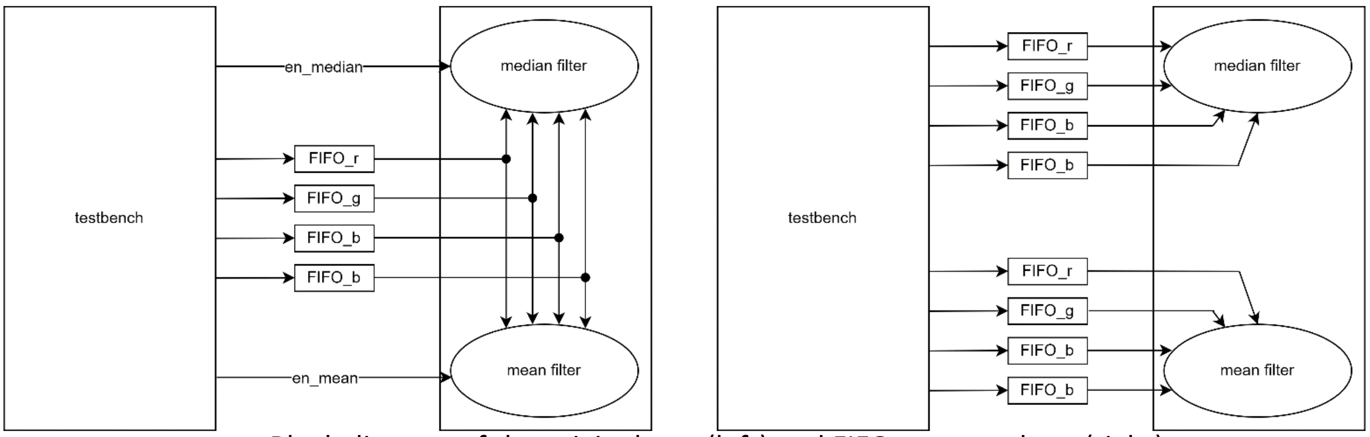
  I this part, we are asked to connect a TLM bus in between initiator and target. I reused the SimpleBus, MemoryMap, and tlm_log in lab4, instantiated the bus in main, and adjust the transport address in testbench.

- Implementation Details

  Part0:

  In my original implementation, the median filter and mean filter takes input from the same FIFO channels. It works fine if I can properly schedule the data read/write from testbench and two filters by calling wait().

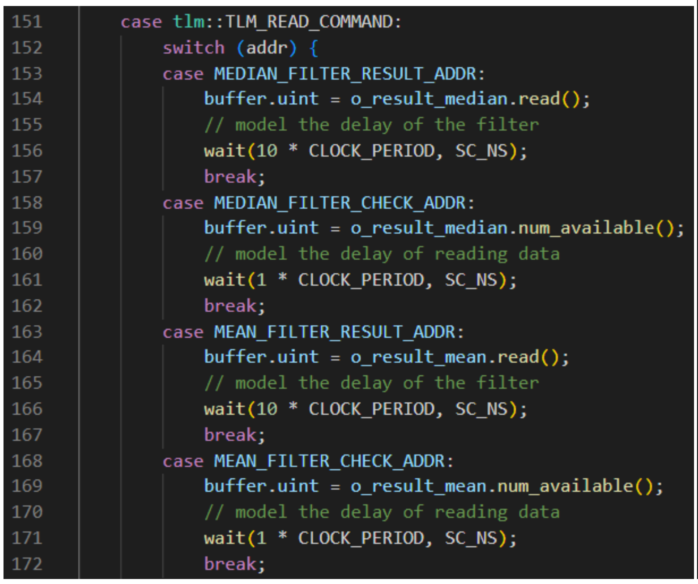
However, with TLM interface and quantum mechanism, we can avoid calling much wait() and hence reduce the simulation time. This cause that we can't control the executing order within a quantum, and the median filter will probably read out additional data, which belongs to mean filter, before being disabled. Then the mean filter will start waiting for data that won't comes anymore, and leads to the suspend of simulation. So, I decided to separate the used FIFO of two filters to avoid this situation, and the two not used enable signals are removed.



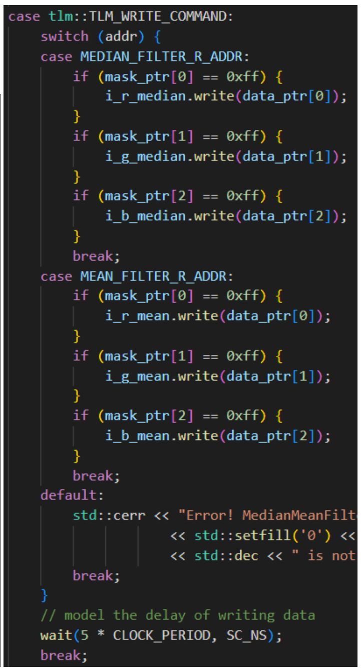
Block diagram of the original one (left) and FIFO separated one(right)
(Other channels are not shown for simplicity)

Part1:

To wrap with TLM interface, I reused the Initiator.h and Initiator.cpp in lab3 as a submodule of the testbench. In the MedianMeanFilter, I change the FIFO ports to FIFO channels, and implemented the blocking transport function. It looks almost the same with the one in SobelFilter except for the read/write address cases are double due to the number of FIFO channel. The wait times I set in blocking transport function are 5 for writing RGB data, 1 for reading the check, and 10 for reading result, which model the computation latency of a filter for one output pixel.



```
    case tlm::TLM_READ_COMMAND:
        switch (addr) {
        case MEDIAN_FILTER_RESULT_ADDR:
            buffer.uint = o_result_median.read();
            // model the delay of the filter
            wait(10 * CLOCK_PERIOD, SC_NS);
            break;
        case MEDIAN_FILTER_CHECK_ADDR:
            buffer.uint = o_result_median.num_available();
            // model the delay of reading data
            wait(1 * CLOCK_PERIOD, SC_NS);
            break;
        case MEAN_FILTER_RESULT_ADDR:
            buffer.uint = o_result_mean.read();
            // model the delay of the filter
            wait(10 * CLOCK_PERIOD, SC_NS);
            break;
        case MEAN_FILTER_CHECK_ADDR:
            buffer.uint = o_result_mean.num_available();
            // model the delay of reading data
            wait(1 * CLOCK_PERIOD, SC_NS);
            break;
```

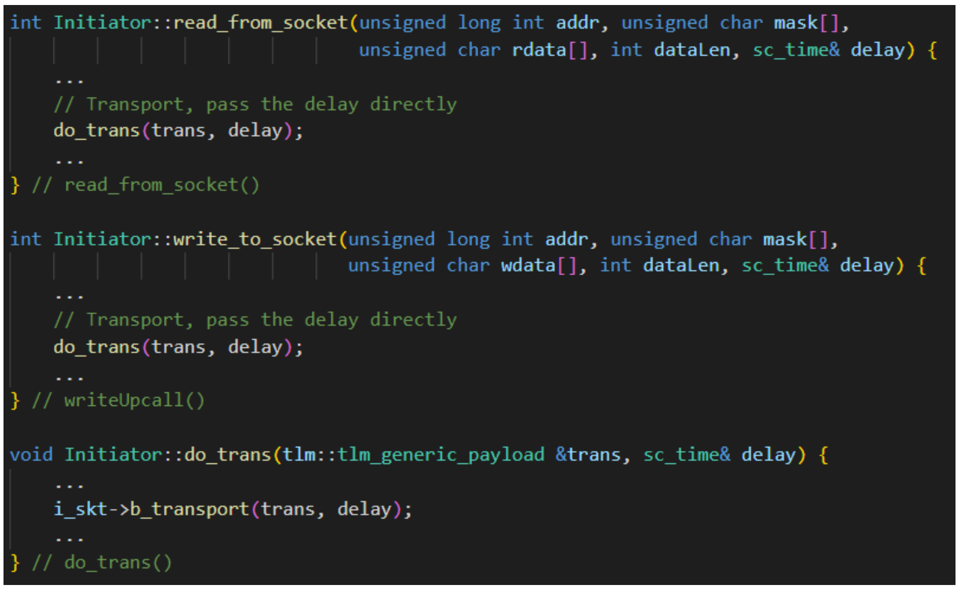


```
case tlm::TLM_WRITE_COMMAND:
    switch (addr) {
    case MEDIAN_FILTER_R_ADDR:
        if (mask_ptr[0] == 0xff) {
            i_r_median.write(data_ptr[0]);
        }
        if (mask_ptr[1] == 0xff) {
            i_g_median.write(data_ptr[1]);
        }
        if (mask_ptr[2] == 0xff) {
            i_b_median.write(data_ptr[2]);
        }
        break;
    case MEAN_FILTER_R_ADDR:
        if (mask_ptr[0] == 0xff) {
            i_r_mean.write(data_ptr[0]);
        }
        if (mask_ptr[1] == 0xff) {
            i_g_mean.write(data_ptr[1]);
        }
        if (mask_ptr[2] == 0xff) {
            i_b_mean.write(data_ptr[2]);
        }
        break;
    default:
        std::cerr << "Error! MedianMeanFilt
                        << std::setfill('0') <<
                        << std::dec << " is not
        break;
    }
    // model the delay of writing data
    wait(5 * CLOCK_PERIOD, SC_NS);
    break;
```

After some modification on testbench, I registered the target blocking transport function and bind the target socket to initiator socket in main. Done.

Part2:

Currently, the Initiator only send dummyDelay (SC_ZERO_TIME) in every transaction. So, I modified the write_to_socket(), read_from_socket(), do_trans() functions to allow the delay being passed to target blocking transport



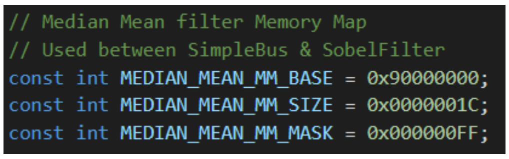
```
int Initiator::read_from_socket(unsigned long int addr, unsigned char mask[],
                                unsigned char rdata[], int dataLen, sc_time& delay) {
    ...
    // Transport, pass the delay directly
    do_trans(trans, delay);
    ...
} // read_from_socket()

int Initiator::write_to_socket(unsigned long int addr, unsigned char mask[],
                               unsigned char wdata[], int dataLen, sc_time& delay) {
    ...
    // Transport, pass the delay directly
    do_trans(trans, delay);
    ...
} // writeUpcall()

void Initiator::do_trans(tlm::tlm_generic_payload &trans, sc_time& delay) {
    ...
    i_skt->b_transport(trans, delay);
    ...
} // do_trans()
```

function.

Then I added a quantum keeper in testbench and tried different quantum value to see the difference of simulation time. This will be discussed in the experiment results part.

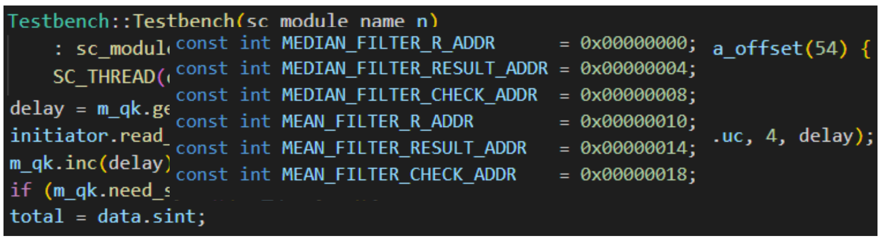
Part3:

To add a bus in between the initiator and target, I reused the SimpleBus, MemoryMap, and tlm_log from lab4. My memory map is defined as follow.



```
// Median Mean filter Memory Map
// Used between SimpleBus & SobelFilter
const int MEDIAN_MEAN_MM_BASE = 0x90000000;
const int MEDIAN_MEAN_MM_SIZE = 0x0000001C;
const int MEDIAN_MEAN_MM_MASK = 0x000000FF;
```

Because I set the local address 0x00~0x0B to median filter, and 0x10~0x1B to mean filter, the MEDIAN_MEAN_MM_SIZE is defined as 0x1C. (local address



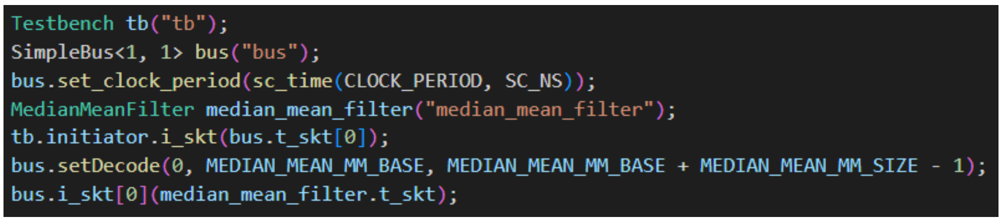
```
Testbench::Testbench(sc module name n)
    : sc_modul const int MEDIAN_FILTER_R_ADDR      = 0x00000000;  a_offset(54) {
    SC_THREAD( const int MEDIAN_FILTER_RESULT_ADDR = 0x00000004;
delay = m_qk.ge const int MEDIAN_FILTER_CHECK_ADDR  = 0x00000008;
               const int MEAN_FILTER_R_ADDR        = 0x00000010;
initiator.read_ const int MEAN_FILTER_RESULT_ADDR  = 0x00000014;  .uc, 4, delay);
m_qk.inc(delay) const int MEAN_FILTER_CHECK_ADDR    = 0x00000018;
if (m_qk.need_s
total = data.sint;
```

0x0C~0x0F are not used)

Then I instantiated the bus in main and connected it to testbench and MedianMeanFilter just like lab4, and adjusted the address for transaction in testbench by adding the module base address. Done.

- Additional Features

By separate the used channels apart, it is possible to process multiple



```
Testbench tb("tb");
SimpleBus<1, 1> bus("bus");
bus.set_clock_period(sc_time(CLOCK_PERIOD, SC_NS));
MedianMeanFilter median_mean_filter("median_mean_filter");
tb.initiator.i_skt(bus.t_skt[0]);
bus.setDecode(0, MEDIAN_MEAN_MM_BASE, MEDIAN_MEAN_MM_BASE + MEDIAN_MEAN_MM_SIZE - 1);
bus.i_skt[0](median_mean_filter.t_skt);
```

images in pipeline. (do mean filter on img1, and do median filter on img2 at the same time)

- Experiment Results

In part2, we are asked to compare the difference of simulation time with and without reducing wait(). To compute the simulation time, I obtain the time before and after simulation by gettimeofday() function. Because the simulation time may vary even in a same setting, I ran the simulation 10 times with same input for each setting.

And in part3, we are asked to count the number of transactions in bus. Note that the number of transactions in the without-qk version indicate the data transfer through FIFO channel, since there is no bus at all.

	Avg. of 10 times (us)	Minimum (us)	Maximum (us)	Num of transactions
Without qk	1,508,387	1,403,494	1,675,807	4,718,592
Quantum = 10	902,650	835,860	973,690	4,718,605
Quantum = 20	995,066	931,071	1,090,419	9,961,475
Quantum = 30	550,798	476,166	613,755	4,718,615
Quantum = 50	862,110	770,074	925,654	15,204,365
Quantum = 100	1,557,661	1,417,627	1,646171	41,418,715

In this chart, we can see that in most of the cases, using quantum mechanism to reduce the number of wait() can actually reduce simulation time.

However, the improvement is not necessarily related to the value of quantum, it relates to your design. In this case, we have several while loops in testbench that check if a result is computed. Because the execution order of processes within a quantum is not pre-defined, the simulation may stuck in the checking loop until a quantum is reached and calling wait(), then the processes can be properly scheduled. With large quantum, the simulation can stuck in a such loop for a long time, and the number of transactions grows dramatically. The quantum=100 version is a good example of this situation.

On the other hand, if the simulation is not stuck much, the simulation time can be highly reduced such as the quantum=10 and quantum=30 cases.

- Discussion and Conclusions

  In this assignment, I learned lots of things including how to wrap a module with TLM interface, the effect of quantum, and the usage of quantum keeper. Being non-familiar with TLM, I wrote codes that make the simulation hangs at first, but after complete this assignment, I have a deeper understanding in systemC and TLM.