Exercise 6: SystemC and Virtual Prototyping

Exercise on TLM Loosely Timed (LT)

Matthias Jung, Lukas Steiner
WS 2022/2023

The source code to start this execise is available here:
`https://github.com/TUK-SCVP/SCVP.Exercise6`

Task 1

Single Initiator and Target

Figure 1 shows an example for a very simple TLM scenario. The code on GitHub for this exercise already provides a simple initiator called `processor`, which fakes the memory access behavior of a simple 32-bit microcontroller by replaying a trace file. As a first step you should study the code of this initiator. Do you recognize the *Regular Expressions* for parsing the input trace? In C++ we have to escape the `\` with a `\`, therefore we see for example patterns like `\\d+` instead of `\d+`.

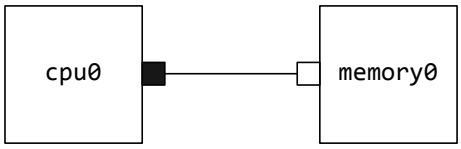

Fig. 1: Initiator and Target

As second part of this task you will implement a simple TLM-LT target called `memory` in the file `memory.h`. The memory size should be provided as a template parameter, and the standard value should be 1024 bytes. If an address greater than or equal to 1024 is accessed, a proper TLM error handling should be done (see lecture) and the simulation should be stopped with `SC_REPORT_FATAL` by the initiator. Hint: Have a look at the artifacts repository

`https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm_lt_initiator_target`

to see an example for an LT-based target. The functions `nb_transport_fw`, `get_direct_mem_ptr` and `transport_dbg` must be implemented as stubs, i.e.,

dummy functions as in the artifacts example. Note that in the target `wait()` should not be called. The time consumed by the memory access (20 ns) should be added to the `delay` variable of the `b_transport` function call.

In order to test your first TLM memory you have to instantiate an object of the class processor (`cpu0`) and an object of the class memory (`memory0`) like this:

```
processor cpu0("cpu0", "stimuli1.txt", sc_time(1, SC_NS));
memory<1024> memory0("memory0");
```

The second constructor parameter for the `cpu0` is the input trace to be read, and the third parameter is the inverse of the frequency, in our case 1 GHz. Then bind the initiator socket of `cpu0` to the target socket of `memory0` in the `main.cpp` file as shown in the lecture.

The `stimuli1.txt` writes some data to the memory in the beginning and reads the same data again some cycles later. If you implemented your memory properly you will see that the data is the same as it was written to the memory before.

Remember to copy the input trace files `stimuli1.txt` and `stimuli2.txt` to the same directory where your executable is located.

You might face the following error when executing your solution:

```
Error: (E549) uncaught exception: regex_error
In file: sc_except.cpp:100
In process: cpu1.process @ 0 s
```

Then uncomment the following line in `processor.h`, rebuild your project and try again.

```
#define GCC_LESS_THAN_4_9_DOES_NOT_SUPPORT_REGEX
```

Task 2

Multiple LT Initiators and Targets

Figure 2 shows an example for a more sophisticated scenario. In this case we have two processors (`cpu0` and `cpu1`), which are connected over an interconnect (`bus0`) to two different memories (`memory0` and `memory1`). The bus should forward each transaction to the right memory based on the memory map shown in Figure 2 (routing).

Implement a bus component in the `bus.h` file and update the `main.cpp` accordingly. `cpu0` should replay the trace file `stimuli1.txt` and `cpu1` the trace file `stimuli2.txt`. Each memory should have a size of 512 bytes.

Hint: Have a look at the artifacts repository

`https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm_lt_initiator_interconnect_target`

again to see an example for an LT-based interconnect with routing.

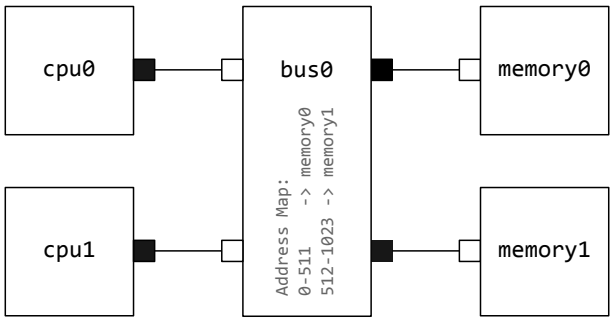

Fig. 2: Initiators, Interconnect and Target

Task 3

Temporal Decoupling

Finally, we will utilize the concept of temporal decoupling to speed up our simulation.

Change the SC_THREAD of the processor from `processTrace` to `processRandom` (inside `processor.h`). This process does not use an input trace file for memory traffic generation but internally generates random time stamps and addresses. In order to achieve a noticeable speedup with temporal decoupling we cannot use input trace files because the file parsing consumes a lot more wall-clock time than the actual system behavior simulation.

In addition, all `cout` statements are removed for a more accurate comparison.

Now modify the processor in such a way that it uses a quantum keeper for temporal decoupling.

Hint: Have a look at the artifacts repository

`https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm_quantum_keeper`

again to see an example for temporal decoupling.

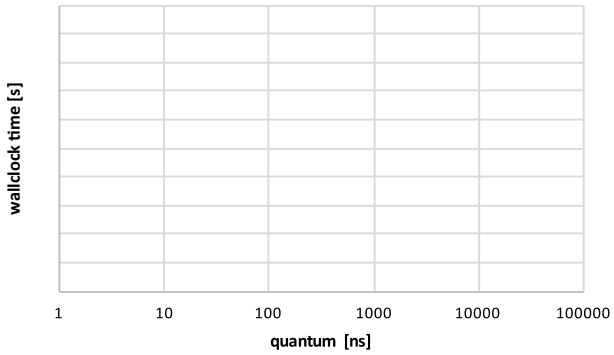

Fig. 3: Quantum vs. Wall-Clock Time

Execute the code with quantums ranging from 1 ns to 100000 ns. For measuring the wall-clock time you can use the `time` command

```
time ./exercise6
```

Put the values into the graph template in Figure 3. You should observe variations in wall-clock time of around 40%.