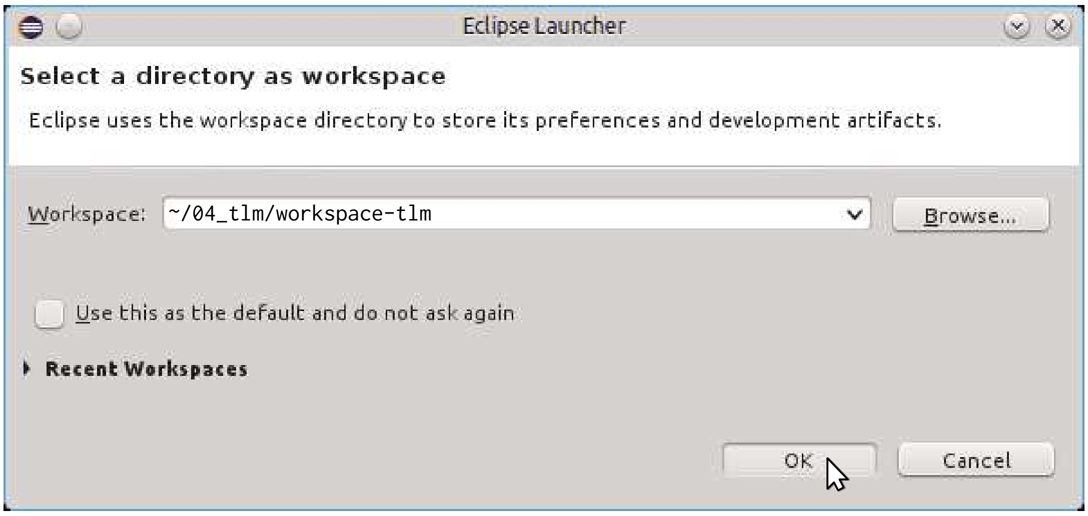
Lehrstuhl für Informatik 12
Cauerstraße 11
91058 Erlangen

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

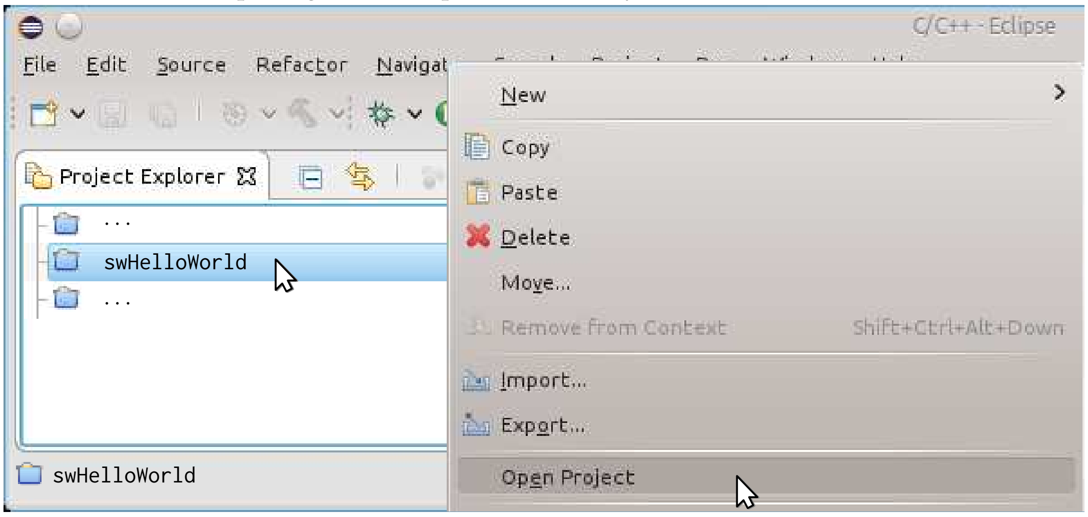
4. Exercise of the Laboratory
Entwicklung interaktiver eingebetteter Systeme

1 Introduction

In the following, you will get familiar with Multi-Processor System-on-Chip (MPSoC) modeling by means of Transaction-Level Modeling (TLM). For this purpose, an eclipse workspace has been prepared for you. Please open the workspace by starting eclipse and choosing the directory depicted in Figure 1a as your workspace directory.


Eclipse Launcher
Select a directory as workspace
Eclipse uses the workspace directory to store its preferences and development artifacts.
Workspace: ~/04_tlm/workspace-tlm
Browse...
Use this as the default and do not ask again
Recent Workspaces
OK
Cancel

(a) Opening the workspace in directory *~/04_tlm/workspace-tlm*



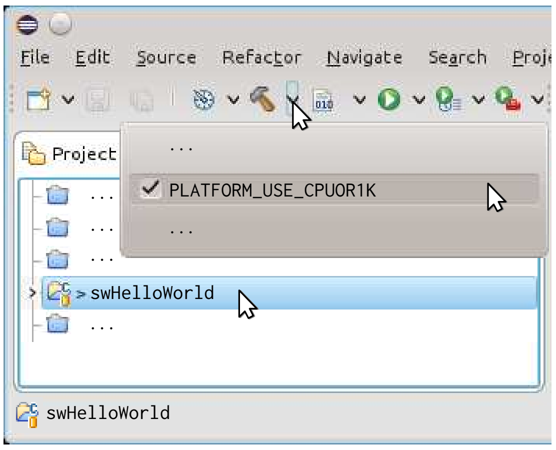
(b) Opening the `swHelloWorld` project in the workspace

Figure 1: The eclipse workspace for the current lab

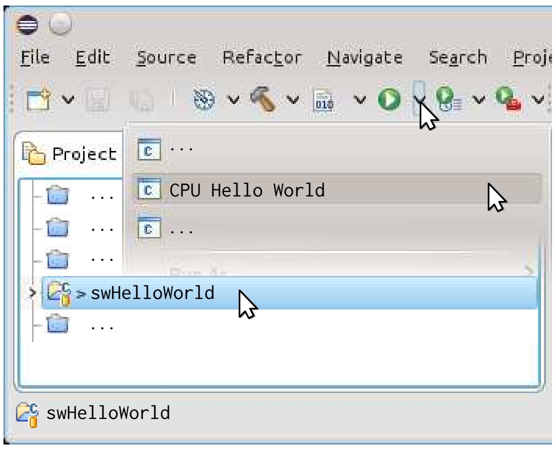
In this workspace, only the hwsim project will be edited by you in the process of completing the lab. However, the following two projects are of relevance for understanding the lab:

1. hwsim – A SystemC simulation environment containing the SystemC module CpuOR1K, which wraps an Instruction Set Simulator (ISS) from Imperas™ for an OpenRISC 1000 CPU, the SimpleMem SystemC module for memory modeling, and, optionally (see Figure 5a), the Bus module when multiple memories should be connected to the CPU.
2. swHelloWorld – A simple "Hello World" program that should be executed on the simulated CPU system provided by the hwsim simulator.

The goal of the current lab is to run a "Hello World" program on the TLM-based MPSoC simulator hwsim. First, open the swHelloWorld project (see Figure 1b) and compile the program (see Figure 2a). This results in the executable *~/04_tlm/workspace-tlm/swHelloWorld/obj/src/main/main.elf*, which should be loaded by the hwsim simulator for simulation with the OpenRISC 1000 CPU. The simulation could be started by selecting the CPU Hello World run target (see Figure 2b). However, this will fail because the hwsim simulator is not yet compiled and, in fact, there is still stuff for you to do to get it working. Moreover, you can find a more detailed documentation about TLM in the PDF files *TLM_2_0_user_manual.pdf* and *TLM_2_0_presentation.pdf* in the directory *~/04_tlm/documentation*.



(a) Compiling the "Hello World"-program



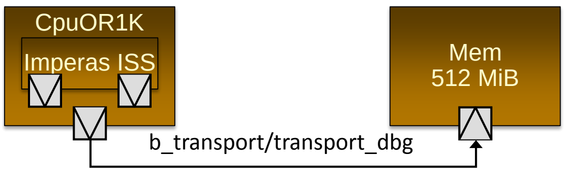
(b) Trying to run the "Hello World"-program

Figure 2: Steps needed to simulated the "Hello World"-program with the TLM-based MPSoC simulator hwsim

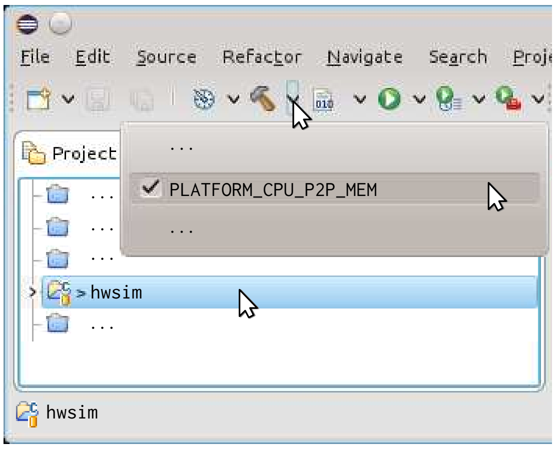
Task 1 (CPU with Dedicated Memory Architecture)

Project:	`hwsim`
Files:	*src/hwsim/cpp/Top.hpp*
	src/hwsim/cpp/Top.cpp
	src/hwsim/cpp/CpuOR1K.hpp
	src/hwsim/cpp/CpuOR1K.cpp
	src/hwsim/headers/SimpleMem.hpp
	src/hwsim/cpp/SimpleMem.cpp

In the following, you will try to realize the architecture depicted in Figure 3 and run the "Hello World" program on the resulting simulator. First, you will realize the architecture of the system by opening the `hwsim` project and editing the corresponding source files. In this phase, the modules will still be skeletons, but they will be correctly connected to each other.



(a) Visualization of the architecture



(b) Compiling the `hwsim` simulator

Figure 3: Simple architecture connecting the CPU directly to the memory

a) You begin by modifying the memory module to have the required constructor and TLM socket to connect it to the rest of the system. Thus, add a constructor for the `SimpleMem` module that takes as parameters the *module name* `name`, the *address of the last addressable byte of the memory* `max_addr`, the `readBandwidth` of type `double` giving the *read bandwidth in MegaBytes (MBs) per second*, and, finally, the `writeBandwidth` of type `double` giving the *write bandwidth in MBs per second*. If no read and write bandwidth are specified, these should default to an infinite bandwidth. You might use a negative value, e.g., -1, to denote this case and then use special handling in `b_transport` to detect it. Also, add a TLM target socket named `m_memory_socket` to the `SimpleMem` module. Take care to also name the socket accordingly in the constructor.

b) Subsequently, you will modify the `CpuOR1K` module to have the required TLM socket. Thus, add a TLM initiator socket named `INSTRUCTION_DATA` to the `CpuOR1K` module. Take care to also name the socket accordingly in the constructor.

c) Finally, the architecture of the system will be specified in the `Top` module. For this purpose, add one instance called `cpu` of the SystemC module type `CpuOR1K` and one instance called `mem` of SystemC module type `SimpleMem` to the `Top` module. The instantiated memory should have a size of 512 MebiByte (MiB). The CPU should get the executable file to be simulated on it, i.e., "*obj/src/main/main.elf*". Take care to also name these instances accordingly in the constructor of the `Top` module and to connect the TLM initiator socket `INSTRUCTION_DATA` from the CPU with the TLM target socket `m_memory_socket` from the memory. For the `parent` parameter of the `CpuOR1K` module, use the `platform` module, which is already instantiated by the `Top` module.

Next, you should try to compile the `hwsim` simulator as seen in Figure 3b. The simulator should compile, however, it will of course not work. Thus, we will continue by implementing the functionality of the memory module.

d) Add dummy methods, i.e., empty code, for the TLM blocking transport interface to the `SimpleMem` class. Do this both for the *plain version* (`b_transport`) as well as the *debug version* (`transport_dbg`) of this interface. Take care to also register these methods with the TLM socket `m_memory_socket` in the constructor via usage of its methods `register_b_transport` and `register_transport_dbg`.

e) Implement `b_transport` by calling `transport_dbg` and adding additional *timing functionality* to handle the timing given to the constructor by the *read and write bandwidth* parameters. Remember that `transport_dbg` simply realizes the functionality without any consideration to timing. To realize the timing functionality, you have to calculate the delay for the transaction and add the calculated delay to the `sc_time &` parameter of the `b_transport` method. The delay in us can be derived by dividing the length (in bytes) of a transaction by the read or write bandwidth given in MBs per second. Don't forget to handle the special case of infinite bandwidth.

f) To store the contents of the memory, you should add an appropriate member variable called `data` to the `SimpleMem` class. Take care to initialize this memory array with the endlessly repeating, i.e., up to the end address of the memory, sequence of `0xDE 0xAD 0xBE 0xEF`.

g) For the functionality of the *debug version*, you should start by writing code to check that the memory access is not out of bounds. That is to say, check that the address of the last accessed byte is not greater than the *address of the last addressable byte of the memory*, which has been given to the constructor. In case of error, set the appropriate TLM error response, e.g., one of the statuses provided by the `tlm::tlm_response_status` enum type.

h) Next, you should handle the `tlm::TLM_READ_COMMAND` command by reading the appropriates bytes from the `data` member variable and storing them in the buffer given by the *data pointer of the generic payload*.

i) Finally, handle the `tlm::TLM_WRITE_COMMAND` command by writing the appropriates bytes to the `data` member variable from the buffer given by the *data pointer of the generic payload*.

Next, we have to tackle the problem that the wrapped ISS from Imperas has two TLM initiator sockets, i.e., `INSTRUCTION.socket` and `DATA.socket`. Both of them have to be forwarded to the `INSTRUCTION_DATA` TLM initiator socket you have added previously.

j) To connect both initiator sockets from the ISS to the `INSTRUCTION_DATA` socket, you will create two new TLM target sockets named `INSTRUCTION_TARGET` and `DATA_TARGET`. Take care to also name these TLM sockets accordingly in the constructor.

k) Add methods for the TLM blocking transport interface to the `CpuOR1K` class. These methods should simply forward the corresponding call to the `INSTRUCTION_DATA` socket, i.e., `INSTRUCTION_DATA->b_transport(...)` and `INSTRUCTION_DATA->transport_dbg(...)`.

l) Register these methods with the `INSTRUCTION_TARGET` and `DATA_TARGET` TLM sockets.

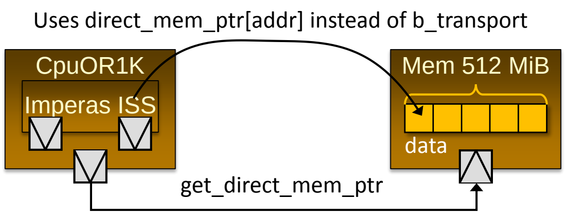
m) Subsequently, connect the sockets `INSTRUCTION.socket` and `DATA.socket` from the Imperas™ OpenRISC 1000 ISS to your `INSTRUCTION_TARGET` and `DATA_TARGET` TLM sockets.

Finally, start the simulation (cf. Figure 2b) and debug your system until the "Hello World" application outputs an infinite sequence of "Hello world!"s on your console.

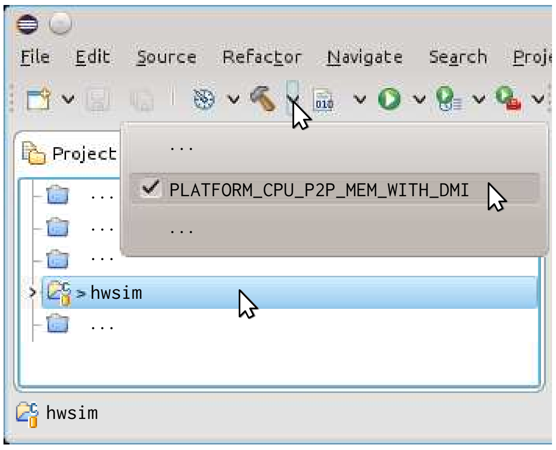
Task 2 (DMI Simulation Acceleration for SimpleMem)

Project:	`hwsim`
Files:	*src/hwsim/cpp/Top.hpp*
	src/hwsim/cpp/Top.cpp
	src/hwsim/cpp/CpuOR1K.hpp
	src/hwsim/cpp/CpuOR1K.cpp
	src/hwsim/headers/SimpleMem.hpp
	src/hwsim/cpp/SimpleMem.cpp

In the following, you will enable support for the Direct Memory Interface (DMI) simulation acceleration technique. The idea behind DMI is that a TLM initiator, e.g., the `CpuOR1K`, can request an `unsigned char` pointer called *direct memory pointer* directly to the *data array* where a TLM target, e.g., the `SimpleMem`, keeps its memory contents, e.g., as depicted in Figure 4a. The TLM interface method to request such a pointer is the method `get_direct_mem_pointer`. However, the TLM initiator is only allowed to use this interface if in a previous transaction, e.g., a call to `b_transport` or `transport_dbg`, the DMI hint in the `tlm_generic_payload` class was turned on by the TLM target.



(a) DMI usage for the simple architecture connecting the CPU directly to the memory


File Edit Source Refactor Navigate Search Proj
Project
...
PLATFORM_CPU_P2P_MEM_WITH_DMI
...
hwsim
hwsim

(b) Compiling the `hwsim` simulator with DMI support

Figure 4: Simple architecture connecting the CPU directly to the memory

To begin, you will extend the `SimpleMem` module to support DMI. For this purpose, the following modifications are necessary:

a) Add an additional constructor to the `SimpleMem` module that takes as parameters the *module name*, the *address of the last addressable byte of the memory*, a DMI flag that indicates if the instantiated memory should support DMI, the `readBandwidth` of type `double` giving the *read bandwidth in MBs per second*, and, finally, the `writeBandwidth` of type `double` giving the *write bandwidth in MBs per second*. If no read and write bandwidth are specified, these should default to an infinite bandwidth. If the DMI flag is `true`, the instantiated memory must support DMI and must not support it otherwise. If this flag is not given, it should default to `false`.

b) Add a dummy method, i.e., empty code, for the TLM `get_direct_mem_ptr` interface to the `SimpleMem` class. Register this method with the TLM socket `m_memory_socket` in the constructors via usage of the method `register_get_direct_mem_ptr`.

c) Modify the methods realizing the blocking transport TLM interface to set the DMI hint flag in their `tlm_generic_payload` argument according to the DMI support status of the instantiated `SimpleMem` module.

d) Implement the dummy method for the TLM `get_direct_mem_ptr` interface method.

Next, you have to modify `CpuOR1K` to also forward the TLM `get_direct_mem_ptr` interface calls from the `INSTRUCTION_TARGET` and `DATA_TARGET` sockets to the `INSTRUCTION_DATA` socket. Moreover, for complete DMI support, there is also the `invalidate_direct_mem_ptr` method belonging to the TLM DMI interface. This method can be called by a TLM target to notify TLM initiators that a previously acquired direct memory pointer is no longer valid. To realize DMI support for the `CpuOR1K` module, the following modifications are necessary:

e) Add a dummy method, i.e., empty code, for the TLM `get_direct_mem_ptr` interface to the `CpuOR1K` class. Take care to also register this method with the `INSTRUCTION_TARGET` and `DATA_TARGET` TLM sockets.

f) Implement the dummy method for the TLM `get_direct_mem_ptr` interface method to forward the call to the `INSTRUCTION_DATA` socket.

g) Add a dummy method, i.e., empty code, for the TLM `invalidate_direct_mem_ptr` interface to the `CpuOR1K` class. Take care to also register this method with the TLM socket `INSTRUCTION_DATA` via usage of the `register_invalidate_direct_mem_ptr` method of the socket.

h) Implement the dummy method for the TLM `invalidate_direct_mem_ptr` interface method to forward the call to both the `INSTRUCTION_TARGET` and `DATA_TARGET` sockets.

The `hwsim` simulator can be compiled in different modes (cf Figures 3b, 4b, and 5b). If DMI should be supported, then `PLATFORM_USE_DMI` will be defined. Thus, you will modify the `Top` module to respect this define.

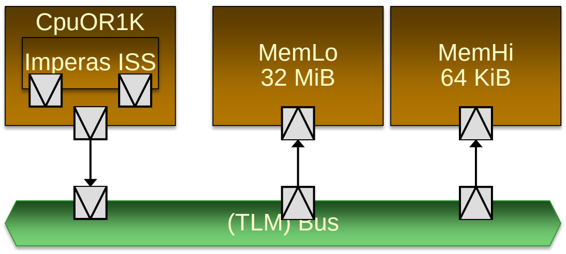
i) Modify the `Top` module to turn on DMI for the `SimpleMem` instance `mem` when the `hwsim` simulator is compiled with DMI support, i.e., when `PLATFORM_USE_DMI` is defined.

j) Run the "Hello World" application on the simulator with and without DMI support and compare the simulation performance.

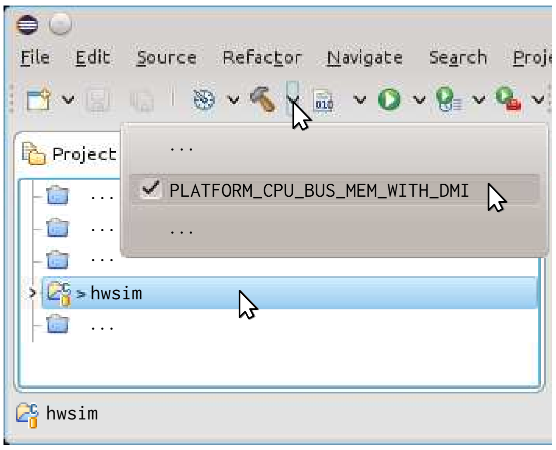
Task 3 (Bus Modelling)

Project:	`hwsim`
Files:	*src/hwsim/cpp/Top.hpp*
	src/hwsim/cpp/Top.cpp
	src/hwsim/cpp/Bus.hpp
	src/hwsim/cpp/Bus.cpp

In the following, we want to model a system with reduced memory requirements. Here, you should take advantage of the fact that the 512 MiB memory is only used at the beginning, i.e., starting from `0x00000000`, for code, data, and heap as well as at the end, i.e., ending at `0x1FFFFFFF`, for the stack. Thus, you will place the memory `memLo` of size 32 MiB at the beginning starting at `0x00000000` and the memory `memHi` of size 64 KiB at the end ending at `0x1FFFFFFF`. That a bus should be used will be indicated by the `PLATFORM_USE_BUS` define that will be set if the `hwsim` simulator is compiled as indicated in Figure 5b.



(a) Visualization of the architecture



PLATFORM_CPU_BUS_MEM_WITH_DMI

(b) Compiling the `hwsim` simulator to use a bus and DMI support

Figure 5: Architecture using a bus to connect the two memories `memLo` and `memHi` to the CPU

In detail, the bus has the two public methods (i) the `getTargetSocket()` method, which should be used to acquire a TLM target socket to which a TLM initiator socket of a bus master, e.g., the `CpuOR1K` CPU, should be connected, and (ii) the `getInitiatorSocket(uint32_t start_range, uint32_t end_range)` method, which should be used acquire a TLM initiator socket to which a TLM target socket of a bus slave, e.g., a `SimpleMem` memory module, should be connected. The connected bus slave should be visible in the address range from `start_range` to `end_range`, i.e., the first addressable byte of the bus slave should be `start_range` and the last addressable byte should be `end_range`. For each forward of a transaction from a TLM target socket (connected to a bus master) to a TLM initiator socket (connected to a bus slave), address rewriting as well as a search for the right TLM initiator socket has to be performed. Address rewriting is necessary as a TLM initiator socket is visible in the address range `start_range`...`end_range`, while the address range of a `SimpleMem` module is `0x0`...`max_addr`. In more detail, the following modifications are necessary for the `Bus` module:

a) You should realize a data structure to store the TLM target sockets acquired via usage of the `getTargetSocket()` method.

b) Implement the `getTargetSocket()` method.

c) You should realize a data structure to store the TLM initiator sockets and their associated address ranges.

d) Implement the `getInitiatorSocket(startRange, endRange)` method.

e) Add dummy methods for the TLM blocking transport and DMI interfaces, i.e, methods for `b_transport`, `transport_dbg`, `get_direct_mem_ptr`, and `invalidate_direct_mem_ptr`.

f) You must also register these methods with the sockets acquired by the `getTargetSocket` and `getInitiatorSocket` methods.

g) Implement address rewriting and the search for the right TLM initiator socket as a method called `decode` that can be used by the TLM interface methods `b_transport`, `transport_dbg`, and `get_direct_mem_ptr`. Please realize a generic solution that uses the address ranges given to the `getInitiatorSocket` method and does not contain hard coded addresses from the following example, which is only used to illustrate the process. To exemplify, consider the transaction with address `0xFFFF1000` shown in Figure 6 that the CPU initiates at point ①. Cleary, the CPU can initiate transaction for its whole address range of 4 GiB, i.e.,

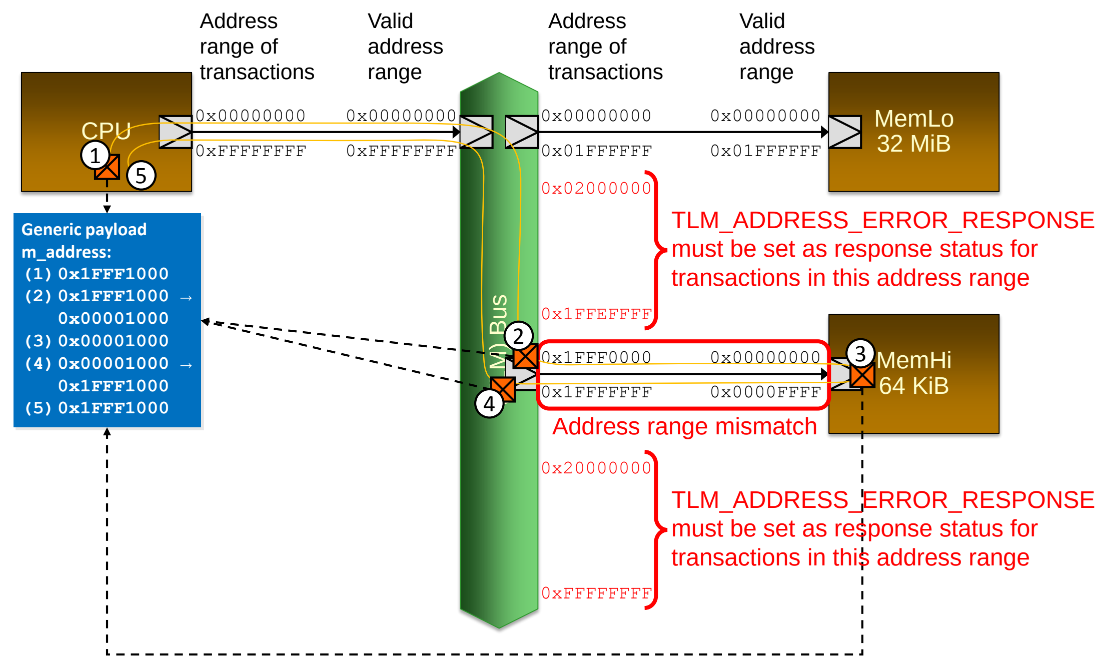


Figure 6: Example of address rewriting for a transaction accessing `memHi`

`0x00000000` to `0xFFFFFFFF`. Moreover, the TLM must also support receiving transaction in this address range. After having received a transaction, the bus must search for the right TLM initiator socket on which to forward the transaction. If none is found, e.g., the address is not between either `0x00000000` to `0x01FFFFFF` (`memLo`) or `0x1FFF0000` to `0x1FFFFFFF` (`memHi`), then the bus must set the `TLM_ADDRESS_ERROR_RESPONSE` status for the transaction. Otherwise, the transaction has to be forwarded to the found TLM initiator socket, e.g., as seen at point ②. However, in general, the address in the transaction has to be modified as otherwise an *address range mismatch* between the address in the transaction and the address range the bus slave is expecting will occur. To exemplify, the address `0xFFFF1000` contained at point ① in the transaction does not fit in the address range `0x00000000` to `0x0000FFFF` expected by `memHi`. Thus, you will have to subtract `0x1FFF0000` from the address in the transaction to modify it in such a way that it will fit into the address ranges expected by the bus slave. Then, the transaction can be forward so that the memory will execute it at point ③.

h) Moreover, in the backward path, i.e., when transactions have been processed by the bus slave connected to the TLM initiator sockets, the address rewriting has to be reversed. For this purpose, you should implement the `encode` method that shifts the address range of the bus slave back into the global address range used by the bus. To exemplify, this happens at point ④ in Figure 6 where `0x1FFF0000` is added to the address to restore it back to its original value. Thus, at point ⑤ the CPU sees a transaction containing the same address as was used to initially send it at point ①.

i) Next, implement the TLM interface methods `b_transport`, `transport_dbg`, as well as the method `get_direct_mem_ptr`. For this methods, you have first to `decode` the address to perform address rewriting and lookup of the correct TLM initiator socket that covers the address range the is requested for the transaction. If no such initiator sockets can be found, be sure to set the corresponding TLM error status for the transaction. Do not forget to reverse the address rewriting after the transaction has been processed by the connected bus slave. Moreover, address rewriting must not only be performed for the `tlm_generic_payload` structure but also for the `tlm_dmi` structure. There, DMI start and end addresses have to be rewritten.

j) Subsequently, the `invalidate_direct_mem_ptr` method should be realized. Here, you will need the `encode` method to shift the address range of the bus slave back into the global address range used by the bus. The `invalidate_direct_mem_ptr` method must be forwarded to all TLM target sockets of the bus, which will notify all bus masters connected to the bus that a DMI pointer these bus masters might have acquired is now invalid.

Finally, you will have to modify the `Top` module to instantiate `memLo`, `memHi`, and `bus` instead of the 512 MiB memory `mem`. Remember, you should only do this if `PLATFORM_USE_BUS` is defined. Also, use the `PLATFORM_USE_DMI` define to decide if DMI should be turned on for `memLo` and `memHi`.

k) Change *Top.hpp* to either have `memLo`, `memHi`, and `bus` if `PLATFORM_USE_BUS` is defined or otherwise keep `mem` if `PLATFORM_USE_BUS` is not defined.

l) Take care to also name these instances accordingly in the constructor of the `Top` module.

m) Next, use the bus methods `getTargetSocket()` and `getInitiatorSocket(...)` to realize the connections of the architecture as depicted in Figure 5a and discussed in the beginning of this tasks description.

Finally, start the simulation (cf. Figure 2b) and debug your system until the "Hello World" application outputs an infinite sequence of "Hello world!"s on your console.