University of Stuttgart
Germany

Institute of Computer Architecture
and Computer Engineering
Dept. of Embedded Systems

Modeling, Simulation, and Specification

Prof. Dr. Martin Radetzki

Exercise 5: *Cycle-Accurate Modelling and Direct Memory Access*

The source files for this exercise are available in the package exercise5.zip on the course webpage.

1. From Architecture View (CX) to Verification View (CA)

   In the directory **bus_ca**, you will find a working solution for problem 3 of exercise 6. Your task is to modify the model so that the cycle-approximate bus (`bus_cx`) is replaced by a cycle-accurate bus `bus_ca`, which is provided in the files **bus_ca.h** and **bus_ca.cpp**. Compile and simulate your result and see what has changed.

   Note: the cycle-accurate bus implements the extended bus interface `ext_bus_if`. This means that all masters must have an `ext_bus_if` port (instead of `bus_if`) and provide an ID as another parameter whenever they initiate a transaction. An ID is already stored in the attribute `id` of the masters; it is set with a constructor parameter when instantiating a master. You will also need to connect a **clock** (use 5 ns cycle time) to the cycle-accurate bus.

2. Adding a direct memory access master

   The implementation of the addition, as developed in the previous exercise is quite inefficient: each addition requires a master to read two operands from memory, write them to the adder (via the adapter), then to read back the result and store it in the memory; a total of 6 transactions as shown in Figure 1.

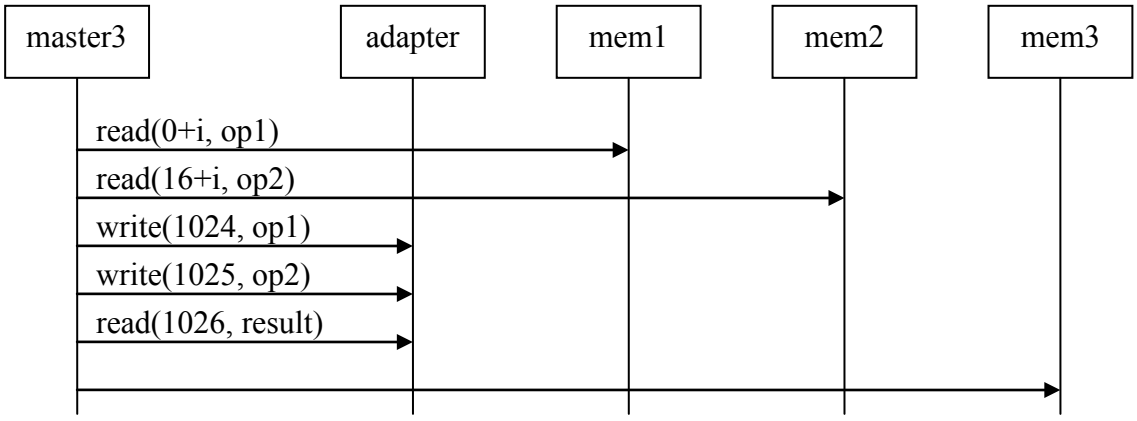

Figure 1

   You will now implement an advanced adapter for the adder that is capable of fetching the addition operands from memory and writing the result to the target memory on its own, without help from the master. This reduces the transactions per addition to a number of 3 (see Figure 2). This is called direct memory access (DMA).

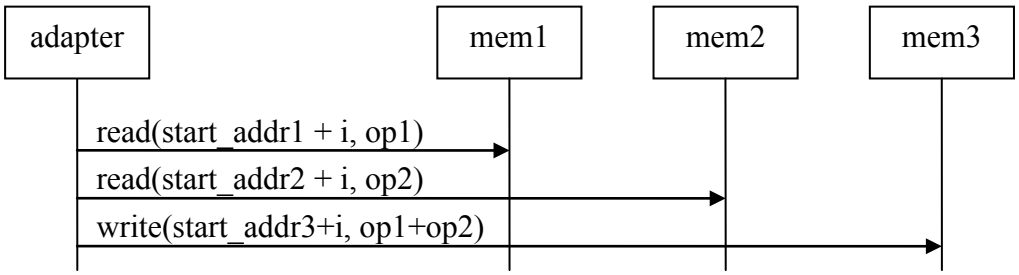

Figure 2

a) In order to gain direct memory access, the adapter will have to become a bus master too. For this purpose, add a suitable port to the adapter and connect it to the bus.

b) The adapter will still remain a slave (in addition to being a bus master). This allows another master (master3 in this example) to program the adapter with the range of memory addresses it shall operate on. Change the read and write methods of the adapter so that the following data can be programmed:
   - **`write(1024, data)`** stores the start address for the first memory in a new attribute **`start_addr1`**,
   - **`write(1025, data)`** stores the start address for the second memory in a new attribute **`start_addr2`**,
   - **`write(1026, data)`** stores the start address for the third memory (target memory) in a new attribute **`start_addr3`**,
   - **`write(1027, data)`** stores the number of additions to be performed in a new attribute **`block_size`**,
   - **`write(1028, data)`** starts the addition for the whole range of values if `data != 0`, the value of **`data`** shall be stored in a new attribute **`status`**,
   - **`read(addr, data)`** returns the values of the above attributes if **`addr`** is in the range of 1024 to 1027, and the status of the addition if the address is 1028, with a 0 indicating that the additions are finished, and other values indicate that it's not finished yet.

c) Add a process (`SC_THREAD`) to the adapter. The process shall be triggered when `write(1028, data)` occurs with `data != 0`, e.g. by introducing an `sc_event`. Then it shall perform the addition of a block of operands as shown by the message sequence chart in Figure 2, for all `i` from `0` to `block_size - 1`. The addition of the two operands shall be performed by the adder (**NOT** in the adapter itself!). When finished, the process shall reset `status` to 0.

d) Change **master3** so that it only programs the adapter with correct start addresses and the size of the data blocks in memory, initiates the addition by writing to address 1028, and queries the status occasionally by reading address 1028 and checking the result for 0.

e) When addition is finished, the master shall stop the simulation.

f) The system has one more master now, and the additional master should use a unique ID to make arbitration work correctly, Make the necessary changes.

g) Compile and run the simulation. Make sure the results in **`mem3`** are still correct. The addition operations should be finished sooner now.