ECSE 541 Assignment 2

Murray Kornelsen
260714814

bus_master_if & bus_minion_if

In my implementation, the bus class implements both the master and minion interfaces, allowing connections through sc_port<bus_master_if> and sc_port<bus_minion_if> interfaces.

Requests are handled by saving requests in a queue associated with each master. The request queue is implemented as a std::vector of std::deque of bus_request pointers. The bus_request struct contains the master id, address, operation, and length. The bus also keeps track of the current request.

```
std::vector<std::deque<bus_request *>> request_queue;
bus_request *cur_request;
```

When the bus is idle (NULL current request), it selects a request from one of the queues in a round robin order. The listen function is then able to return information about the current request. Once a minion has received a request to its address space, it can call the acknowledge function which sets a boolean flag and triggers an event. The waitForAcknowledge function waits until that flag is set and the current request matches the waiting master.

To synchronize the read and write functions between the master and minion, I define booleans bus_ready and data_ready. For a read, the master sets bus_ready to true and then waits until data_ready becomes true. The minion waits until bus_ready is true, indicating that it should put data onto the bus, at which point it sets the shared variable bus_data, sets data_ready to true, and sets bus_ready to false. The master, having seen that data_ready became true, sets its output variable and then sets data_ready to false (ensuring that bus_ready and data_ready are both false before the next operation). The event rq_complete_event is then triggered and both master and minion wait for the signal ready_for_next_event. These events are used by the bus to update its internal state so the next operation is handled correctly.

The write process is very similar but with the roles of master and minion switched. For a write, the minion has to set bus_ready while the master waits. The master then puts data on the bus, sets data_ready and unsets bus_ready. Finally, the minion accepts the data, updates the current request, and notifies the bus of the request completion.

The various request types are defined in the bus_if header. I define four request types: single read, single write, burst read, and burst write. In order to determine when a request is complete and the next request in the queue should be accepted, the current request length is decremented upon the completion of each read or write. For single requests, it is set to 0 on read or write completion.

Performance Modelling

In order to capture the performance of the bus, I decided to create a version of the bus that uses a clock signal to synchronize its actions. Perhaps counterintuitively, I think this version is actually simpler than the untimed version, as instead of needing signals to indicate each bus state change, the processes can simply wait for the next clock edge and then check the needed variable. Using this approach, I can then add the specified delays to the bus operations by adding the appropriate number of wait statements. For example, to ensure that a request takes two clock cycles, it is implemented as follows:

```
void Request(unsigned int mst_id, unsigned int addr, unsigned i
    wait(clk.posedge_event());
    wait(clk.posedge_event());
    bus_request *req = new bus_request(mst_id, addr, op, len);
    request_queue.at(mst_id).push_back(req);
}
```

For the read and write operations that require communication through boolean flags, waiting can be achieved simply as in this example from the ReadData function:

```
while (!data_ready) {
    wait(clk.posedge_event());
}
```

Performance Breakdown

In order to analyze the performance of the hardware, software, and bus models, I created a few versions of the clocked matrix multiplication application. The first version is a software implementation that only uses single reads and writes. This emulates a processor that has a very limited register file and can only store a single 32 bit number in each. As such, the processor needs to load values one at a time before performing the multiplication and addition. This is extremely slow as the software cannot take advantage of burst reads and, although I didn't model extra delays, the arithmetic operations would likely take longer than specialized hardware.

The second implementation uses a hardware module that handles one location of the output at a time by loading a row of a and a column of b before computing the dot product. This allows two optimizations: using a burst read to fetch rows and only fetching each column once. In this implementation, the software uses a burst write to the hardware units address space. It provides the hardware with the row and column of the location in the output to be computed. When the computation is complete, a flag is set which can be read by a single read by the software. The software polls this flag (using the loop shown in the following image) as it waits for the hardware to finish. Once that flag is set, the output value can be read by the software from the hardware and then written into memory.

```
tmp = 0;
while (!tmp) {
    bus_addr = HW_ADDR + HW_OUT_READY_POS;
    bus_op = OP_SINGLE_READ;
    bus_len = 1;
    sw_bus_master->Request(bus_mst_id, bus_addr, bus_op, bus_len);
    sw_bus_master->WaitForAcknowledge(bus_mst_id);
    sw_bus_master->ReadData(tmp);
    cout << "sw - done = " << tmp << endl;
}
```

The final implementation uses a more extensive hardware module. In this implementation, the hardware unit is supplied with the input and output addresses of the matrices. The hardware unit then reads both matrices entirely using one burst read for each. The matrices are stored in large buffers and the matrix product is computed instantly. Finally, the hardware directly writes the result back into memory using a burst write and sets a flag as in the previous implementation. The software uses the same polling loop as the previous method to wait for the hardware completion. This implementation saves a lot of time through the use of burst operations, removing a lot of time spent issuing requests and handling acknowledgements. A lot of time is also saved on handling the software polling the hardware completion flag, which cannot be interleaved between operations of burst requests.

The total time taken to compute the matrix product and write to memory for each implementation is shown in the following table.

Implementation	Computation time (us)
Software only	21.354531
Row/Col hardware	15.355131
Full matrix hardware	2.609739

As we can see from the table, being able to use burst reads and preventing memory locations from being read multiple times improve the time taken by a huge amount. If it were not for the polling requests, the row/col approach would have improved over the software only approach by an even larger margin. The third approach, which optimizes the matrix reading and writing, subsequently reduces the time spent by nearly 90% compared to the software only approach.