Homework #5 Solution

Problem 1)

Our memory controller is configured for a DRAM with 4 banks. To get the same behavior as in homework 2 problem 2, we'll add two variables to the mem class for each back. The first variable is a flag that indicates whether or not any address has been accessed, to see if an ACTIVE command-latency needs to be modeled. The second variable stores the last address accessed, to see if both ACTIVE and PRECHARGE command-latencies need to be modeled. Since there are 4 banks, these can be added as length-4 arrays. Following is the code that was added to mem.h (highlighted in red):

```
class mem: public sc_core::sc_module
{
  public:

  mem( sc_core::sc_module_name module_name,
       sc_dt::uint64  memory_size  // memory size (bytes)
      );

  tlm_utils::simple_target_socket<mem>  slave;

  private:

  bool m_initialized[4];
  sc_dt::uint64 m_memory_size, m_last_addr[4];

  void custom_b_transport
  ( tlm::tlm_generic_payload &gp, sc_core::sc_time &delay );

};
```

Following is the modified code from mem.cpp. If a transaction is larger than 64 bytes (one burst-16 transaction), then it is broken into multiple burst-16 transactions (not necessary for this problem, but makes the solution more general). Following the solution for Homework 2 problem 2, each burst-4 transaction required 4 cycles plus a certain amount of control overhead:

Xact 2: 10 cycles = 4 cycles (for each bus transfer) + 6 cycles (control overhead)
Xact 3: 15 cycles = 4 cycles (for each bus transfer) + 11 cycles (control overhead)
Xact 4: 12 cycles = 4 cycles (for each bus transfer) + 8 cycles (control overhead)

Note that we have used the simulated control overheads, rather than the calculated control overheads, which were 4 cycles fewer in the Homework 2 Problem 2 solution. Ignoring the first transaction, we can assume that the second transaction requires only 10 cycles, because the row is already active. When the new row is accessed in the 3rd transaction, then an additional 15 cycles are needed, because the current row must be precharged before activating the new row. When a new bank is accessed in the 4th transaction, 12 cycles are needed, because the new row must be activated, but there is no currently opened row to be precharged. This additional latency is added into the memory delay for each case. Note that the code has been added for the READ command only, because the problem did not request that WRITE commands be modeled.

```
SC_HAS_PROCESS(mem);
mem::mem( sc_core::sc_module_name module_name,sc_dt::uint64 memory_size )
  : sc_module (module_name)
  , m_memory_size (memory_size)
{
  slave.register_b_transport(this, &mem::custom_b_transport);
  for ( int i=0 ; i<4 ; i++ )
    m_initialized[i]=false;
}

void
mem::custom_b_transport
 ( tlm::tlm_generic_payload &gp, sc_core::sc_time &delay )
{
  sc_dt::uint64    address   = gp.get_address();
  tlm::tlm_command command   = gp.get_command();
  unsigned long    length    = gp.get_data_length();
  unsigned long    last_burst_length,bank;
  sc_core::sc_time mem_delay(10,sc_core::SC_NS);

  wait(delay);
  bank=(unsigned long)((address & 0x0000000000006000)>>13);
  if (address < m_memory_size) {
    switch (command) {
      case tlm::TLM_WRITE_COMMAND:
      {
...
      }
      case tlm::TLM_READ_COMMAND:
      {
         last_burst_length = (length % 0x40)/4;
        // Break transaction into bursts of length 16
        for (unsigned long i=0; i < length; i+=0x40) {
          // Initialize delay to number of cycles in burst
          if (i+0x40 >= length)  // Last burst
            mem_delay=sc_core::sc_time(10*last_burst_length,
                                       sc_core::SC_NS);
          else // Burst 16
            mem_delay=sc_core::sc_time(160,sc_core::SC_NS);
          // Add additional control latency
          if (!m_initialized[bank]) {
            mem_delay=mem_delay+sc_core::sc_time(80,sc_core::SC_NS);
            m_initialized[bank]=true;
          }
          else if ((address & 0xFFFF8000) ==
                   (m_last_addr[bank] & 0xFFFF8000))
            // Same Row
            mem_delay=mem_delay+sc_core::sc_time(60,sc_core::SC_NS);
          else
            // New Row
            mem_delay=mem_delay+sc_core::sc_time(110,sc_core::SC_NS);
          wait(mem_delay);
          cout << sc_core::sc_time_stamp() << " " << sc_object::name();
          cout << " READ 0x" << hex << address << endl;
          m_last_addr[bank]=address;
          address+=0x40;
        }
        gp.set_response_status( tlm::TLM_OK_RESPONSE );
        break;
      }
      default:
...
```

The code for top.cpp is given below. The size of the memory must be increased to something large enough to allow accesses to different rows.

```
top::top(sc_core::sc_module_name name)
  : sc_core::sc_module(name)
  , bus("bus")
  , mem0("mem0", 0x100000)
  , mem1("mem1", 0x100000)
  , stub0("stub0","xact0.txt")
  , stub1("stub1","xact1.txt")

{
  stub0.master(bus.target_socket[0]);
  stub1.master(bus.target_socket[1]);
  bus.initiator_socket[0](mem0.slave);
  bus.initiator_socket[1](mem1.slave);
}
```

The code for xact0.txt is given below. Four 16-byte transactions are listed, all starting at time 2000, so that all delays will match those in Homework 2 problem 2(d). xact1.txt is blank, so as not to confuse the simulation.

```
# time command bytes address
2000 ns READ   0x10   0x00000000
2000 ns READ   0x10   0x00001000
2000 ns READ   0x10   0x00008000
2000 ns READ   0x10   0x00002000
```

The simulation output is given below, showing 4 bursts and an increase in delay when the row boundary is crossed. Comparing these delays to the behavior in Homework 2 problem 2, each transaction completes at nearly the same time.

```
0 s top.stub1 Completed
2 us top.stub0 READ 10 0
2 us top.mem02120 ns top.mem0 READ 0x0
2120 ns top.stub0 READ 10 1000
2120 ns top.mem02220 ns top.mem0 READ 0x1000
2220 ns top.stub0 READ 10 8000
2220 ns top.mem02370 ns top.mem0 READ 0x8000
2370 ns top.stub0 READ 10 2000
2370 ns top.mem02490 ns top.mem0 READ 0x2000
2490 ns top.stub0 Completed
```

Problem 2)

The basic approach to this solution is to add the following:

- a peq_with_get (payload event queue) to store transactions as they come in, so that they can be handled sequentially
- an array of events for each initiator (called event_done[]) to indicate when the transaction is complete. The b_transport call waits on each event.
- a map to associate the appropriate event_done[] element with each transaction
- a thread-process to wait on the PEQ, process each transaction, and notify the appropriate event_done[] element when complete.

The initiatorBTransport function is then modified to put the payload in the PEQ, set the event_done[] element in the map, and wait for the event to be notified.

The code for the modified SimpleBusLT.h is given below, with changes in red.

```
#include "tlm.h"

#include "tlm_utils/simple_target_socket.h"
#include "tlm_utils/simple_initiator_socket.h"
#include "tlm_utils/peq_with_get.h"
#include <map>

template <int NR_OF_INITIATORS, int NR_OF_TARGETS>
class SimpleBusLT : public sc_core::sc_module
{
...
public:
  target_socket_type target_socket[NR_OF_INITIATORS];
  initiator_socket_type initiator_socket[NR_OF_TARGETS];

  sc_core::sc_event event_done[NR_OF_INITIATORS];
  std::map<transaction_type*,sc_core::sc_event*> event_map;

public:
  SC_HAS_PROCESS(SimpleBusLT);
  SimpleBusLT(sc_core::sc_module_name name) :
    sc_core::sc_module(name), m_peq("peq")
  {
...
    for (unsigned int i = 0; i < NR_OF_TARGETS; ++i) {
      initiator_socket[i].register_invalidate_direct_mem_ptr(this,
            &SimpleBusLT::invalidateDMIPointers, i);
    }

    SC_THREAD(main);
  }
...
```

(continued on next page)

Changes to SimpleBusLT.h (continued)

```
  tlm_utils::peq_with_get<transaction_type> m_peq;

  void main(void)
  {
    transaction_type *gpp;
    sc_core::sc_time t(0,sc_core::SC_NS);

    while (true) {
      wait(m_peq.get_event());
      gpp=m_peq.get_next_transaction();
      while (gpp) {
        initiator_socket_type* decodeSocket;
        unsigned int portId = decode(gpp->get_address());
        assert(portId < NR_OF_TARGETS);
        decodeSocket = &initiator_socket[portId];
        gpp->set_address(gpp->get_address() & getAddressMask(portId));
        (*decodeSocket)->b_transport(*gpp, t);
        event_map[gpp]->notify();
        gpp=m_peq.get_next_transaction();
      }
    }
  }
...
  void initiatorBTransport(int SocketId,
                           transaction_type& trans,
                           sc_core::sc_time& t)
  {
    event_map[&trans]=&event_done[SocketId];
    m_peq.notify(trans,t);
    wait(event_done[SocketId]);
  }
...
};
```

The first few lines of the output are shown below. Note that events initiated simultaneously are now executed by the memories sequentially.

```
0 s top.stub0 WRITE 4 10000200
0 s top.stub1 WRITE 4 100
10 ns top.mem1 WRITE 0x200
20 ns top.mem0 WRITE 0x100
40 ns top.stub0 WRITE 4 10000204
40 ns top.stub1 WRITE 4 104
50 ns top.mem1 WRITE 0x204
60 ns top.mem0 WRITE 0x104
80 ns top.stub0 READ 4 10000200
80 ns top.stub1 READ 4 100
90 ns top.mem1 READ 0x200
```

Problem 3)

Running the fibonacci and gregoryleibniz programs with both Spike and the emulator, we get the following output:

```
$ make sim
time spike --isa=rv64gc -l fibonacci.riscv 2> fibonacci.spike.out
starting...
fib(15) = 610
mcycle = 25129
minstret = 25135

real    0m0.706s
user    0m0.206s
sys     0m0.476s
```

```
$ make vsim
time emulator-freechips.rocketchip.system-DefaultConfig +max-
cycles=100000000 +verbose fibonacci.riscv 3>&1 1>&2 2>&3 | spike-
dasm > fibonacci.emulator.out
starting...
fib(15) = 610
mcycle = 28219
minstret = 25333

real    0m18.439s
user    0m36.035s
sys     0m4.687s
```

```
$ make sim
time spike --isa=rv64gc -l gregoryleibniz.riscv 2>
gregoryleibniz.spike.out
starting...
pi(100)*1000 = 3146
mcycle = 818
minstret = 823

real    0m0.278s
user    0m0.089s
sys     0m0.178s
```

```
$ make vsim
time emulator-freechips.rocketchip.system-DefaultConfig +max-
cycles=100000000 +verbose gregoryleibniz.riscv 3>&1 1>&2 2>&3 |
spike-dasm > gregoryleibniz.emulator.out
starting...
pi(100)*1000 = 3146
mcycle = 24950
minstret = 1031

real    0m18.796s
user    0m36.626s
sys     0m4.834s
```

Pulling the output into a spread-sheet, we can calculate the required values as follows:

	fibonacci		gregoryleibniz	
	sim	vsim	sim	vsim
mcycle	25,129	28,219	818	24,950
minstret	25,135	25,333	823	1,031
time (s)	0.706	18.439	0.278	18.796
cps perf	35,593.48	1,530.398	2,942.446	1,327.41
CPI	1.000239	0.897728	1.006112	0.041323

As we would expect, the CPI for the ISS is close to 1, and the cycles-per-second performance is higher for the ISS than for the emulator (23 times faster for the Fibonacci program). What's most surprising is that the number of cycles with the gregoryleibniz program differs so significantly between the ISS and emulator executions. This is most likely due to the fact that the gregoryleibniz.c program uses floating-point instructions. It seems that Spike is assuming that these instructions are executed in 1 cycle each, while the Rocket-Chip requires many more cycles. Examining gregoryleibniz.emulator.out, you can see that nearly 18,000 are devoted to executing the "fdiv" instruction alone.