NC State University
ECE Department　　　　ECE 720　　　　W. Rhett Davis

Homework #5 Solution

Problem 1)

First, calculate the command latencies for the SDRAM, assuming a 100 MHz clock rate (10 ns period):

Latency	Time (ns)	Cycles w/ 100 MHz clock
CAS/CL	-	2
RCD	15	2
RP	15	2
RAS	40	4
RC	55	6
RRD	10	2
WTR	-	2
WR	14	2

Next, determine how long each transaction should take. With 16 data bits (2 bytes) on the DRAM and bursts of length 8, a 64-byte read should take 4 burst-8 read transactions. Each burst read should take 4 cycles plus the CAS latency (6 cycles total). Each time a new row is accessed, the number of cycles should be 4 (each cycle of the burst) + 2 (CAS latency) + 2 (RP command latency) + 2 (RCD command latency) = 10 cycles.

For an 8 Mbit DRAM, given a byte-addressable space, the total size of the address space is 0x100000. Given that the number of column bits is 8 and that 1 address bit is needed to compute the byte offset in one DRAM word, each group of 0x000200 contiguous bytes resides in a separate row in the DRAM. We'll choose the address for the first read to be 0x050000 and the second read to 0x050200 to show the 10-cycle delay to a read in a different row.

To implement this behavior, we'll add a variable to the mem class to store the last address accessed, as follows (from mem.h):

```
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:

  sc_dt::uint64 m_memory_size, m_last_addr;

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

};
```

Following is the modified code from mem.cpp. Each transaction is broken into 32-byte bursts, with the appropriate delay, depending on the address.

```
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();

  sc_core::sc_time mem_delay(10,sc_core::SC_NS);

  wait(delay);
  if (address < m_memory_size) {
    switch (command) {
      case tlm::TLM_WRITE_COMMAND:
      {
        for (unsigned long i=0; i < length; i+=0x20) {
          if ( (address & 0xFFFFFE00) == (m_last_addr & 0xFFFFFE00) )
            mem_delay=sc_core::sc_time(60,sc_core::SC_NS);
          else
            mem_delay=sc_core::sc_time(100,sc_core::SC_NS);
          wait(mem_delay);
          cout << sc_core::sc_time_stamp() << " " << sc_object::name();
          cout << " WRITE 0x" << hex << address << endl;
          m_last_addr=address;
          address+=0x20;
        }
        gp.set_response_status( tlm::TLM_OK_RESPONSE );
        break;
      }
      case tlm::TLM_READ_COMMAND:
      {
        for (unsigned long i=0; i < length; i+=0x20) {
          if ( (address & 0xFFFFFC00) == (m_last_addr & 0xFFFFFC00) )
            mem_delay=sc_core::sc_time(60,sc_core::SC_NS);
          else
            mem_delay=sc_core::sc_time(100,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=address;
          address+=0x20;
        }
        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 match the size of the DRAM.

```
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. Two 64-byte transactions are listed, both starting at time 0, so that all delays will be due to the mem module (no delay from the stub). xact1.txt is blank, so as not to confuse the simulation.

```
# time command bytes address
  0 ns WRITE   0x40   0x00050000
  0 ns WRITE   0x40   0x00050200
```

The simulation output is given below, showing 4 bursts and an increase in delay when the row boundary is crossed.

```
0 s top.stub0 WRITE 40 50000
0 s top.stub1 Completed
100 ns top.mem0 WRITE 0x50000
160 ns top.mem0 WRITE 0x50020
160 ns top.stub0 WRITE 40 50200
260 ns top.mem0 WRITE 0x50200
320 ns top.mem0 WRITE 0x50220
320 ns top.stub0 Completed
```

Problem 2)

To measure the CPI and cycles-per-second performance of the RTL simulation, the XACT_DUMP macro is defined in the test-bench, and the Fibonacci iterations is increased to 21, which yields

```
...
#             12320325 READ addr=000000f0 size=word data=a012b510
#             12320335: Simulation stop requested by CPU
#
# ** Note: Data structure takes 7135080 bytes of memory
#          Process time 50.19 seconds
#          $finish    : cortexm0ds_tb.v(240)
#    Time: 12320335 ns  Iteration: 1  Instance: /cortexm0ds_tb
```

RTL performance = 1,232,033/50.19 = 24,547 cycles/sec

It's actually quite a bit faster when XACT_DUMP is not defined. I get 41,721 cycles/sec in that case. The number above also shows the performance when running on local disk space. Running on network disk-space will slow the simulation considerably, if the XACT_DUMP process is used.

The following is the output of the addr_stat.py script:

```
STATISTICS for addr range 0x0 to 0xffff
Number of Reads:           471441
Number of Writes:          42
Minimum address accessed:  0x0
Maximum address accessed:  0x468
```

The number of reads to this range is a fairly good estimate of the number of instructions executed (assuming that each read is an instruction-fetch). Therefore, the CPI can be calculated as follows:

CPI=1,232,033/471,441 = 2.61

For the TLM simulation, the same approach is used, but we don't need to use the XACT_DUMP process. Also, we'll increase the increasing the Fibonacci iterations to 24 to get interesting output. Here's the output:.

```
...
fib(23) = 28657
finishing...
SystemC: simulation stopped by user.
Simulation Time: 46535350 ns
STATISTICS for addr range 0x0 to 0xffff
Number of Blocking Reads:  2203825
Number of Blocking Writes: 15
Minimum address accessed:  0x0
Maximum address accessed:  0x8454
...
24.68user 5.40system 0:30.20elapsed 99%CPU (0avgtext+0avgdata
6422576maxresident)k
0inputs+0outputs (0major+2361492minor)pagefaults 0swaps
```

RTL performance = 4,653,535/30.2 = 154,090 cycles/sec

Using the same approach as above to calculate CPI, we get

CPI = 4,653,535/2,203,825 = 2.11

Problem 3)

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
```