NC State University
ECE Department ECE 720 W. Rhett Davis

Homework #2 Solution

Problem 1)

First, note the command latencies for the SDRAM, taken from the "Memory Controller Parameters" on the course web-page:

Latency	Cycles w/ 100 MHz clock
CAS/CL	2
RCD	2
RP	3
RAS	5
RC	7
WR	2

With 16 data bits (2 bytes) on the DRAM, a 16-byte transaction would be a burst of length 8, which takes 4 cycles in a DDR memory.

Adding the CAS latency, we would expect a burst to take 4+2=6 cycles total.

If an ACTIVE command must be issued to open a new row, then the number of cycles should be 4 (each cycle of the burst) + 2 (CAS latency) + 2 (RCD command latency) = 8 cycles.

If an PRECHARGE command must also be issued to close an active row, then the number of cycles should be 4 (each cycle of the burst) + 2 (CAS latency) + 2 (RCD command latency) + 3 (RP command latency) = 11 cycles.

According to the Memory Controller Parameters posted on the course web-page, the column address width is 12, and the number of data-bits is 16 (i.e. each address references 2 bytes). Therefore, each active row within a bank has an address space that is 12+1=13 bits, i.e. address bits [12:0]. Because there are 4 banks and this memory uses row-high addressing, we know that address bits [14:13] can be used to select the bank.

The following transaction file is used to simulate the latencies:

```
# time command bytes address data
2000 READ   0x10  0x00000000 0x00
2000 READ   0x10  0x00001000 0x00
2000 READ   0x10  0x00008000 0x00
2000 READ   0x10  0x00002000 0x00
```

Running the simulation with this input file gives the following results:

```
#                  1005: Memory Controller leaving startup...
#                  1015: Simulation leaving reset
#                  2105 READ addr=00000000 size=word data=00020000
#                  2115 SEQ READ addr=00000004 size=word data=00020000
#                  2125 SEQ READ addr=00000008 size=word data=00020000
#                  2135 SEQ READ addr=0000000c size=word data=00020000
#                  2205 READ addr=00000010 size=word data=00000000
#                  2215 SEQ READ addr=00000014 size=word data=00000000
#                  2225 SEQ READ addr=00000018 size=word data=00000000
#                  2235 SEQ READ addr=0000001c size=word data=00000000
#                  2355 READ addr=00010000 size=word data=00000000
#                  2365 SEQ READ addr=00010004 size=word data=00000000
#                  2375 SEQ READ addr=00010008 size=word data=00000000
#                  2385 SEQ READ addr=0001000c size=word data=00000000
#                  2475 READ addr=00004000 size=word data=00000000
#                  2485 SEQ READ addr=00004004 size=word data=00000000
#                  2495 SEQ READ addr=00004008 size=word data=00000000
#                  2505 SEQ READ addr=0000400c size=word data=zzzzzzzz
#                  2515: Simulation stop requested by CPU
```

The state of the memory controller is unknown for the first transaction, and so it's difficult to predict the latency. But it should be possible to predict the latency for transactions 2, 3, & 4. Also, we can assume that each subsequent transaction begins on the last cycle of the previous transaction.

Transaction 2)

Address 0x00001000 lies within the same 13-bit address space as the first transaction (0x00000000), so we know that it is within the same bank and row as the first transaction.

Because this transaction does not require an ACTIVE or PRECHARGE command, we know that it should take 6 cycles, as calculated above.

The simulated time for this transaction is (2235-2135)/10 = 10 cycles, which is 4 cycles more than expected.

Transaction 3)

Address 0x00008000 keeps the bank bits the same but changes address bit 15, which is a row-address bit. Thus, it should require activation of a new row.

This transaction will require both ACTIVE and PRECHARGE commands, and so we know that it should take 11 cycles.

The simulated time for this transaction is (2385-2235)/10 = 15 cycles, 4 more than expected.

Transaction 4)

Address 0x00002000 changes address bit 13, which is a bank address bit, which means accessing a new bank.

Because this bank is not currently active, no PRECHARGE command needs to be issued. Only an ACTIVE command is needed to open the row. This means that 8 cycles should be needed for the transaction.

The simulated time for this transaction is (2505-2385)/10 = 12 cycles, 4 more than expected.

In summary every transaction appears to take 4 cycles more than expected. This may be due to additional logic delays inside the memory controller, or may be due to protocol latencies that we haven't included.

Note that your simulated latencies might differ from the latencies here. The DW_memctl module is quite complex and difficult to predict at times.

Problem 2)

The following boxes show the modifications to each file that was provided, with the changes in bold, red type.

stub.h, stub.cpp, hw04p1.cpp: no change, other than to comment-out the “cout” statement to reduce the volume of output.

SimpleBusLT.h: added member variables to count transactions, code to initialize variables in the constructor, code to increment variables in transport method, a new method to print out the result after 10 us, and registered process in the constructor.

```
using namespace std;
. . .
public:
  target_socket_type target_socket[NR_OF_INITIATORS];
  initiator_socket_type initiator_socket[NR_OF_TARGETS];
  unsigned long bytes_read;
  unsigned long bytes_written;
public:
  SC_HAS_PROCESS(SimpleBusLT);
  SimpleBusLT(sc_core::sc_module_name name) :
    sc_core::sc_module(name),bytes_read(0),bytes_written(0)
  {
. . .
    SC_THREAD(main);
  }

void main(void)
  {
    wait(sc_core::sc_time(10000,sc_core::SC_NS));
    cout << sc_core::sc_time_stamp() << " " << sc_object::name() << endl;
    cout << "  " << bytes_read/0.00001 << " bytes read per sec" << endl;
    cout << "  " << bytes_written/0.00001 << " bytes written per sec" << endl;
  }
. . .

void initiatorBTransport(int SocketId,
        transaction_type& trans,sc_core::sc_time& t)
  {
. . .
    (*decodeSocket)->b_transport(trans, t);

    if (trans.get_command()==tlm::TLM_WRITE_COMMAND)
      bytes_written+=trans.get_data_length();
    if (trans.get_command()==tlm::TLM_READ_COMMAND)
      bytes_read+=trans.get_data_length();
```

Problem 2) (continued)

```
tlm_utils::simple_target_socket<mem>  slave;

void main(void);

private:

sc_dt::uint64 m_memory_size;
unsigned long bytes_read;
unsigned long bytes_written;

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

mem.h: member variables added to record the number of bytes written & read, as well as a process to do the counting.

Problem 2) (continued)

mem.cpp: Initialization of count variables & registration of process in constructor, implementation of process to count reads & writes, commented out "cout" lines in transport method, and added code to increment counts.

```
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)
  , bytes_read(0), bytes_written(0)
{
  slave.register_b_transport(this, &mem::custom_b_transport);
  SC_THREAD(main);
}

void mem::main(void)
{
  wait(sc_core::sc_time(10000,sc_core::SC_NS));
  cout << sc_core::sc_time_stamp() << " " << sc_object::name() << endl;
  cout << "  " << bytes_read/0.00001 << " bytes read per sec" << endl;
  cout << "  " << bytes_written/0.00001 << " bytes written per sec" << endl;
}

void
mem::custom_b_transport
 ( tlm::tlm_generic_payload &gp, sc_core::sc_time &delay )
{
. . .
  wait(delay+mem_delay);
  //cout << sc_core::sc_time_stamp() << " " << sc_object::name();
  if (address < m_memory_size) {
    switch (command) {
      case tlm::TLM_WRITE_COMMAND:
      {
        //cout << " WRITE 0x" << hex << address << endl;
        gp.set_response_status( tlm::TLM_OK_RESPONSE );
        bytes_written+=gp.get_data_length();
        break;
      }
      case tlm::TLM_READ_COMMAND:
      {
        //cout << " READ 0x" << hex << address << endl;
        gp.set_response_status( tlm::TLM_OK_RESPONSE );
        bytes_read+=gp.get_data_length();
        break;
      }
```

Problem 2) (continued)

top.h: Increased the number of initiators

```
class top : public sc_core::sc_module
{
public:

  top (sc_core::sc_module_name name);

  private:

  SimpleBusLT<3, 2> bus;
  mem mem0;
  mem mem1;
  stub stub0;
  stub stub1;
  stub stub2;
};
```

top.cpp: Initialized the new initiator and bound its socket

```
top::top(sc_core::sc_module_name name)
  : sc_core::sc_module(name)
  , bus("bus")
  , mem0("mem0", 4*1024)
  , mem1("mem1", 4*1024)
  , stub0("stub0","xact0.txt")
  , stub1("stub1","xact1.txt")
  , stub2("stub2","xact2.txt")

{
  stub0.master(bus.target_socket[0]);
  stub1.master(bus.target_socket[1]);
  stub2.master(bus.target_socket[2]);
  bus.initiator_socket[0](mem0.slave);
  bus.initiator_socket[1](mem1.slave);
}
```

The output of the simulation is given below:

```
10 us top.mem1
  3.081e+08 bytes read per sec
  2.937e+08 bytes written per sec
10 us top.bus
  5.745e+08 bytes read per sec
  5.504e+08 bytes written per sec
10 us top.mem0
  2.664e+08 bytes read per sec
  2.567e+08 bytes written per sec
10027 ns top.stub2 Completed
10031 ns top.stub1 Completed
10302 ns top.stub0 Completed
Simulation process exited.
```

Problem 3) The code to implement this behavior is given below

CPU.h

```
#pragma once
#include <tlm.h>

class CPU: public sc_core::sc_module
  , virtual public tlm::tlm_bw_transport_if<>
{

public:
  tlm::tlm_initiator_socket<> master;

  SC_HAS_PROCESS(CPU);
  CPU(sc_core::sc_module_name name);

  void main ();

private:

  // Not implemented, but required by interface
  void invalidate_direct_mem_ptr
  (sc_dt::uint64 start_range, sc_dt::uint64 end_range)
  { return; }

  // Not implemented, but required by interface
  tlm::tlm_sync_enum nb_transport_bw

  (tlm::tlm_generic_payload &payload,
   tlm::tlm_phase &phase,
   sc_core::sc_time &delta
  )
  { return tlm::TLM_ACCEPTED; }

};
```

Problem 3) (continued)

CPU.cpp

```
#include "CPU.h"

using namespace std;

CPU::CPU( sc_core::sc_module_name name) : sc_module( name )
 {
    master(*this);
    SC_THREAD(main);
 }

void CPU::main( )
{
  sc_core::sc_time delay(0,sc_core::SC_NS);

  tlm::tlm_generic_payload  *transaction_ptr;
  unsigned char             *data_buffer_ptr;

  transaction_ptr = new tlm::tlm_generic_payload();
  data_buffer_ptr = new unsigned char[1];
  transaction_ptr->set_data_ptr ( data_buffer_ptr );

  sc_dt::uint64 mem_address;
  transaction_ptr->set_command          ( tlm::TLM_READ_COMMAND );
  transaction_ptr->set_data_length      ( 1                     );
  transaction_ptr->set_streaming_width  ( 1                     );

  while (true) {
    mem_address = 0;
    transaction_ptr->set_address          ( mem_address                   );
    transaction_ptr->set_response_status  ( tlm::TLM_INCOMPLETE_RESPONSE );
    master->b_transport(*transaction_ptr, delay);
       cout << sc_core::sc_time_stamp().to_string() << " CPU read "
            << (int)(*data_buffer_ptr) << " from address 0\n";
       wait(20, sc_core::SC_NS);
    if (*data_buffer_ptr) {
      mem_address = 1;
      transaction_ptr->set_address          ( mem_address                   );
      transaction_ptr->set_response_status  ( tlm::TLM_INCOMPLETE_RESPONSE );
      master->b_transport(*transaction_ptr, delay);
         cout << sc_core::sc_time_stamp().to_string() << " CPU read "
           << (*data_buffer_ptr) << " from address 1\n";
         wait(20, sc_core::SC_NS);
       }
  }
}
```

Problem 3) (continued)

UART.h

```
#pragma once

#include <tlm.h>
#include "tlm_utils/simple_target_socket.h"
#include <list>

class UART: public sc_core::sc_module
//, virtual public tlm::tlm_fw_transport_if<>   /// inherit from TLM "forward
interface"

{
public:

  tlm_utils::simple_target_socket<UART>  slave;
  std::list<char> buffer;
  UART(sc_core::sc_module_name module_name);
  void main ();

private:

  void custom_b_transport
  ( tlm::tlm_generic_payload  &payload
  , sc_core::sc_time          &delay_time
  );

  public:
};
```

Problem 3) (continued)

UART.cpp

```
#include "UART.h"
using namespace  std;

SC_HAS_PROCESS(UART);
UART::UART( sc_core::sc_module_name module_name)
    : sc_module (module_name)
{
  slave.register_b_transport(this, &UART::custom_b_transport);
  SC_THREAD(main);
}

void UART::custom_b_transport
( tlm::tlm_generic_payload  &payload
, sc_core::sc_time          &delay_time
)
{
    sc_dt::uint64 addr = payload.get_address();
    unsigned char *data = payload.get_data_ptr();

    if (addr == 0) {
         if (buffer.size()>0) data[0]=(char)1;
         else data[0]=(char)0;
       }
       else {
      if (buffer.size()>0) {
           data[0]=buffer.front();
           buffer.pop_front();
         }
         else data[0]=(char)0;
       }
    payload.set_response_status(tlm::TLM_OK_RESPONSE);
       return;
}

void UART::main()
{
    const char *str = "Hello, World!\n";
       const char *p = str;

    while (true) {
         cout << sc_core::sc_time_stamp().to_string() << " UART receiving " << *p <<
endl;
         buffer.push_back(*p++);
         if (!*p) p=str;
         wait(100, sc_core::SC_NS);
    }
}
```

Problem 3) (continued)

top.h

```
#pragma once

#include <tlm.h>
#include "CPU.h"
#include "UART.h"

class top : public sc_core::sc_module
{
  private:
  CPU   cpu_inst;
  UART  uart_inst;

  public:
  top( sc_core::sc_module_name name): sc_core::sc_module(name),
     cpu_inst("cpu0"), uart_inst("uart0")
  {
    cpu_inst.master(uart_inst.slave);
  }
};
```

hw4p4.cpp

```
#include "top.h"
#include <tlm.h>

int sc_main(int argc, char *argv[])
{
  top top_inst("top0");
  sc_core::sc_start(1000,sc_core::SC_NS);
  return 0;
}
```

The output of the simulation looks identical to the output of problem 1 on homework 5.