

Lecture 6

Transaction Level Modeling in SystemC

Multimedia Architecture and Processing Laboratory

多媒體架構與處理實驗室

Prof. Wen-Hsiao Peng (彭文孝)

pawn@mail.si2lab.org

2007 Spring Term

Acknowledgements

This lecture note is partly contributed by Prof. Gwo Giun Lee (李國君) in the Dept. of EE, National Cheng-Kung University and his team members 王明俊, 林和源 in the research laboratory 多媒體系統晶片實驗室

E-mail: clee@mail.ncku.edu.tw

Tel: +886-6-275-7575 ext. 62448

Web: <http://140.116.216.53>

Reference

- ◆ T. Grotker, S. Liao, G. Martin, S. Swan, "*System Design with SystemC*", Kluwer Academic, 2004 (ISBN: 1-4020-7072-1)

Transaction Level Modeling

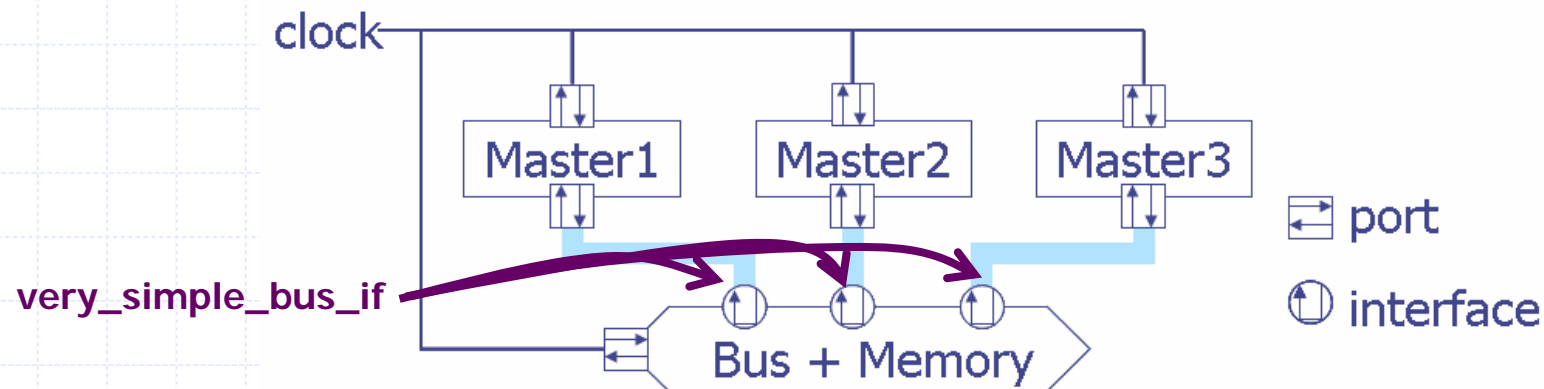
- ◆ A high-level approach to model **digital** systems
 - ❖ Care **more** on what data are transferred to and from what locations
 - ❖ Care **less** on the actual protocol used for data transfer
- ◆ Features
 - ❖ Details of communication are separated from details of computation
 - ❖ Communication mechanisms are modeled as channels
 - ❖ Low-level details of information exchanged are hidden in channels
 - ❖ Pin level details at the structural boundary are abstracted into interface
 - ❖ Transaction requests take place by calling interface functions
 - ❖ Synchronization details of channels are typically abstracted into blocking and/or non-blocking I/O
- ◆ Advantages
 - ❖ Enable high simulation speed by hiding **uninteresting** details



Timed TLM – The Very Simple Bus

The Very Simple Bus

- ◆ Although this example is very simple and may not be practical, it provides us an essential concept about TLM in SystemC
- ◆ Some behaviors of the real bus such as arbitration, split transactions, and memory wait states are not considered
- ◆ Memory is modeled as a memory array within the bus rather than a memory module external to the bus
- ◆ Contention, arbitration, interrupts, and cycle-accuracy can be modeled with TLM without resorting to pin-accuracy



Implementation of The Very Simple Bus

```
class very_simple_bus_if : virture public
sc_interface
{
public:
    virtual void burst_write (char *data,
                               unsigned adder, unsigned length)=0;
    virtual void burst_read (char *data,
                              unsigned adder, unsigned length)=0;
};
```

```
class very_simple_bus:
public very_simple_bus_if,
public sc_channel
{
public:
    very_simple_bus ( sc_name nm,
                     unsigned mem_size,
                     sc_time cycle_time )
    : sc_channel(nm), _cycle_time(cycle_time)
    {
        //we model bus memory access using an
        //embedded memory array
        _mem = new char [mun_size];
        //set initail value of memory to zero
        memset(_mem, 0, mem_size_);
    }
    ~very_simple_bus() { delet [] _mem;}
```

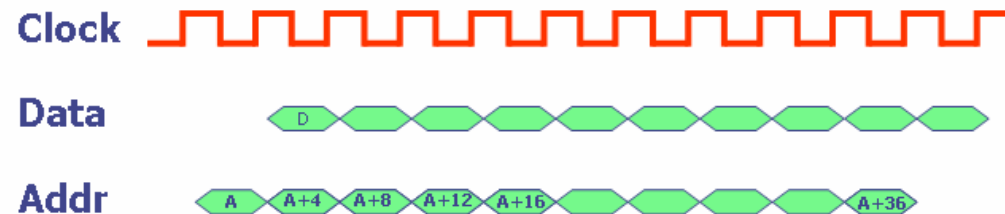
```
virture void burst_read (char *data, unsigned adder, unsigned length)
{
    //model bus contention using mutex, but no arbitration rules
    _bus_mux.lock();
    // block the caller for length of burst transaction
    Wait (length * _cycle_time);
    // copy the data form memory of burst transaction
    memcpy(data, _mem +addr, length);
    // unlock the mutex to allow others access to the bus
    _bus_mutex.unlock();
}
virture void burst_write (char *data, unsigned adder, unsigned length)
{
    _bus_mutex.lock();
    wait (length * _cycle_time);
    // copy the data form requestor to memory
    memcpy(_mem+addr, data, length);
    _bus_mutex.unlock();
}
protected:
char* _mem;
sc_time _cycle_time;
sc_mutex _bus_mutex;
};
```

**Cycle-count
accurate
model**

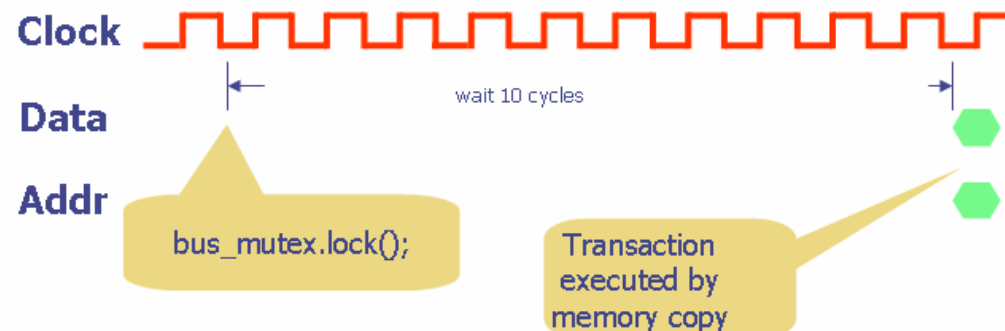
Suppression of Uninteresting Details

- ◆ Burst transfer may take many cycles to complete
- ◆ The bus is merely doing routine works
- ◆ Clients that have pending bus requests are just waiting

RTL



TLM

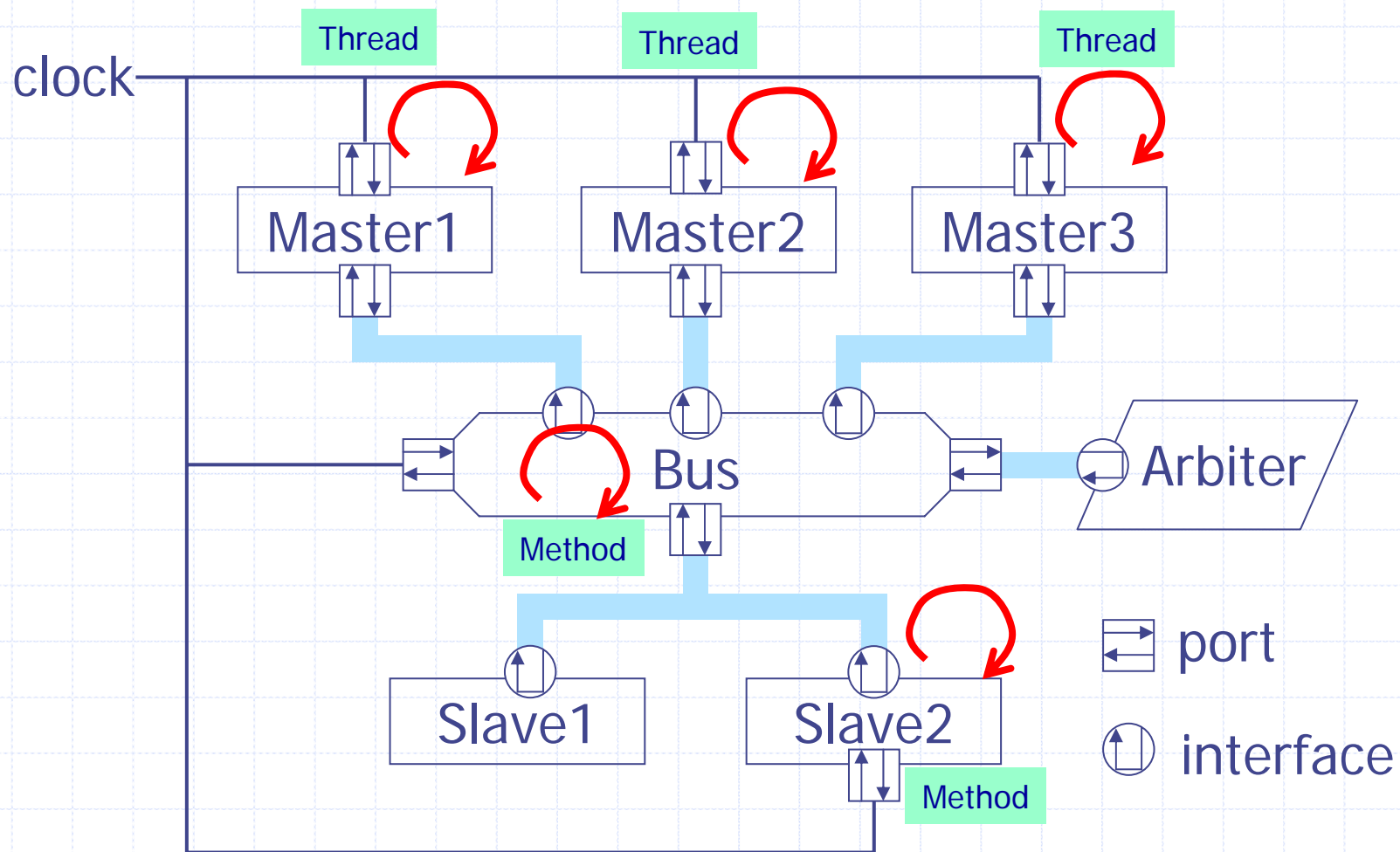


No need to devote simulation time for uninteresting details



Timed TLM – The Simple Bus Design

Simple Bus Design (c. 1)



Structure of the Simple Bus Design

- ◆ Masters (CPUs, DSPs, Arithmetic Intensive ASIC)
 - ❖ Initiate transactions on the bus
- ◆ Bus
 - ❖ Allow the masters and slaves to communicate using bus transactions
- ◆ Slaves (ROMs, RAMs, I/O Devices, Hardware Accelerators)
 - ❖ Response to the bus requests
- ◆ Arbiter
 - ❖ Arbitrate which master can issue the transaction via the bus
 - ❖ Select a request to execute from the competing bus requests
 - ❖ When a master is granted access to the bus, the requests from the other masters are queued by the bus and executed in later cycles
- ◆ Clock generator
 - ❖ Provide the system clock that can synchronize the blocks

Structure of the Simple Bus Design (c. 1)

- ◆ Master 1: Blocking Master
 - ❖ Use **blocking** master interface
 - ❖ Model high-level software that initiates transactions as they execute
- ◆ Master 2: Non-Blocking Master
 - ❖ Use **non-blocking** master interface
 - ❖ Model detailed processor (instruction-set simulator, ISS) that must execute on every clock edge even when waiting for its bus transactions to complete
- ◆ Model Master 3: Direct Master
 - ❖ Use **direct** master interface
 - ❖ Print debug information about the contents of the memories
 - ❖ Does not represent a block that will exist in the real world

Structure of the Simple Bus Design (c. 2)

◆ Slave 1 (Fast Memory)

- ❖ Implement **slave** interface
- ❖ Model a random access memory that supports single-cycle read/write operation with no wait states and no clock port
- ❖ React immediately to the bus request and set the status

◆ Slave 2 (Slow Memory)

- ❖ Implement **slave** interface
- ❖ Model a random access memory which takes a few number of cycles to complete a read/write operation, and contains a clock port

◆ Wait states

- ❖ Additional cycles that a slave takes to complete an operation
- ❖ All other activity on the bus waits until the operation completes

Features of the Simple Bus Design

- ◆ High performance, cycle-accurate, platform transaction-level model
- ◆ Cycle-accurate transaction level modeling
 - ❖ Model is done at transaction level
 - ❖ Model is based on cycle-based synchronization
- ◆ Cycle-based synchronization
 - ❖ Model the data movement on a clock by clock basis
 - ❖ Sub-cycle events are of no interest
- ◆ Transaction-based modeling
 - ❖ Communication between components are described as function calls
 - ❖ Sequences of events on a group of wires are denoted by a set of function calls in an abstract interface
- ◆ Two-phase synchronization
 - ❖ Modules attached to the bus execute on the rising clock edge
 - ❖ The bus executes on a falling clock edge

Features of the Simple Bus Design (c. 1)

- ◆ Easy to add different kinds and numbers of masters or slaves
 - ❖ Masters connect to the bus using just one port connection
 - ❖ Slaves connect to the bus using SystemC *multi-port* feature
- ◆ Easy to change the arbitration policy by replacing the arbiter
 - ❖ Arbiter is a separate module from the bus

Ideas behind the Simple Bus Model

◆ Modeling efforts

- ❖ Relatively easy to develop, understand, use, and extend
- ❖ Capable of being constructed very early in the system design
- ❖ Enable designers to explore implementation alternatives
- ❖ Make design trade-offs before it is too late or too expensive to do so

◆ Accuracy

- ❖ Being fully cycle-accurate
- ❖ Being able to accurately simulate with both the SW and HW components
- ❖ Fast and accurate enough to validate SW before more detailed HW models or implementations are available

◆ Speed

- ❖ Capable of simulate at the speed of more than 0.1MHz
- ❖ Fast enough to allow meaningful amounts of SW to be executed along with HW models



Master Interface

Master Interface

- ◆ Describe the communication between the **master** and the **bus**
 - ❖ Master interface is used by masters and implemented in the bus
- ◆ 3 sets of master interface functions
 - ❖ Blocking master interface
 - ❖ Non-blocking master interface
 - ❖ Direct master interface
- ◆ Multiple masters can be connected to a bus
 - ❖ Each master is independent of the others
 - ✦ Each master can issue a bus request at any time
 - ❖ Each master is identified by a priority number
 - ✦ The lower the priority is, the more important the master is
 - ✦ Each master interface function use this priority to set the importance of the call
 - ❖ A master can reserve the bus for a subsequent access
 - ✦ The bus can be locked for the same master in consecutive cycles

Blocking Master Interface

- ◆ These methods return only after the transaction is completed
- ◆ Used by high-level software that generate read/write transactions
 - ❖ Such software model is not cross-compiled to a target processor and executes directly on the host workstation

```
class simple_bus_blocking_if : public virtual sc_interface
{
public: // blocking BUS interface
virtual simple_bus_status burst_read(unsigned int unique_priority
                                     , int *data
                                     , unsigned int start_address
                                     , unsigned int length = 1
                                     , bool lock = false) = 0;

virtual simple_bus_status burst_write(unsigned int unique_priority
                                      , int *data
                                      , unsigned int start_address
                                      , unsigned int length = 1
                                      , bool lock = false) = 0;
}; // end class simple_bus_blocking_if
```

(1) The id of the master
(2) The importance of the master

If lock is set,
(1) The bus is reserved for exclusive use for a next request of the same master
(2) The function cannot be interrupted by a request with a higher priority

Return Values of Master Interface Methods

◆ SIMPLE_BUS_REQUEST

- ❖ The request is issued and placed in the queue
- ❖ The status in all cases right after issuing the request
- ❖ The status only changes when the bus processes the request

◆ SIMPLE_BUS_WAIT

- ❖ The request is being served but not completed yet

◆ SIMPLE_BUS_OK

- ❖ The request is completed without errors

◆ SIMPLE_BUS_ERROR

- ❖ The request is finished but the transfer is not complete successfully

Non-Blocking Master Interface

- ◆ These functions return immediately, but the read/write will take more than one cycle when competing requests exist
- ◆ Caller must check the status of the last request using `get_status()`
- ◆ Used by ISS models which cannot be suspended while they have outstanding bus requests

```
class simple_bus_non_blocking_if : public virtual sc_interface{  
  
public: // non-blocking BUS interface  
virtual void read (unsigned int unique_priority  
                  , int *data  
                  , unsigned int address  
                  , bool lock = false) = 0;  
  
virtual void write (unsigned int unique_priority  
                   , int *data  
                   , unsigned int address  
                   , bool lock = false) = 0;  
  
virtual simple_bus_status get_status (unsigned int unique_priority) = 0;  
  
}; // end class simple_bus_non_blocking_if
```

Non-Blocking Master Interface (c. 2)

- ◆ A non-blocking request can be made if the status of the last request is either SIMPLE_BUS_OK or SIMPLE_BUS_ERROR
- ◆ An error message is produced and the execution is aborted when a new request is issued and the current one is not completed yet

Direct Master Interface

- ◆ These functions provide instantaneous read/write
 - ❖ Simulated time will not advance and scheduler will not intervene
 - ❖ Data accesses go through the bus for proper routing of the requests
 - ❖ Data transfer is done without using bus protocol
- ◆ Used for creating simulation monitors
 - ❖ Enable debuggers running on top of ISS models to read/write to slaves without waiting for the simulation time to advance

```
class simple_bus_direct_if : public virtual sc_interface
{

public:
    // direct BUS/Slave interface
    virtual bool direct_read(int *data, unsigned int address) = 0;
    virtual bool direct_write(int *data, unsigned int address) = 0;

}; // end class simple_bus_direct_if
```




Slave and Arbiter Interfaces

Slave Interface

- ◆ Describe the communication between the **bus** and the **slave**
 - ❖ Slave interface is used by the bus and implemented by every slave
 - ❖ By definition, the slaves thus play the role of channels
- ◆ 2 sets of slave interface functions
 - ❖ Normal slave interface
 - ✦ Serve the default read/write to and from the slaves
 - ❖ Direct slave interface
 - ✦ Similar to direct master interface
- ◆ Multiple slaves can be connected to a bus
 - ❖ Two functions can be used to obtain the memory range of a slave
 - ❖ `unsigned int start_address() const;`
 - ❖ `unsigned int end_address() const;`

Normal Slave Interface

- ◆ The read/write function performs a single data transfer and returns immediately, and caller must check the return values
- ◆ Return values of slave interface methods
 - ❖ **SIMPLE_BUS_WAIT**: the slave issues a wait state
 - ❖ **SIMPLE_BUS_OK**: the transfer was successful
 - ❖ **SIMPLE_BUS_ERROR**: an error occurs during the transfer
- ◆ If the return status is **SIMPLE_BUS_WAIT**, caller must call the function again until the status becomes **SIMPLE_BUS_OK**

Map requests to the appropriate slave

```
class simple_bus_slave_if : public simple_bus_direct_if
{
public: // Slave interface
    virtual simple_bus_status read(int *data, unsigned int address) = 0;
    virtual simple_bus_status write(int *data, unsigned int address) = 0;
    virtual unsigned int start_address() const = 0;
    virtual unsigned int end_address() const = 0;
}; // end class simple_bus_slave_if
```


Transfer a single data item to or from the slave

Arbiter Interface

- ◆ Describe the communication between the **bus** and the **arbiter**
 - ❖ Arbiter interface is used by the bus and implemented in the arbiter
 - ❖ By definition, the arbiter thus plays the role of channel
- ◆ Arbitrate competing requests issued by different masters
 - ❖ The bus passes its outstanding requests to an arbiter on each cycle
 - ❖ One of the requests is selected for execution based on arbitration policy while the others are kept in the SIMPLE_BUS_REQUEST state

```
class simple_bus_arbiter_if : public virtual sc_interface{  
  
public:  
virtual simple_bus_request* arbitrate(const simple_bus_request_vec &requests) = 0;  
  
}; // end class simple_bus_arbiter_if
```

Outstanding requests are passed
to the arbiter as a vector



Master and Slave Request Status

- ◆ Master request status (read by the master)
 - ❖ SIMPLE_BUS_REQUEST
 - ✦ The request is issued and placed in the queue
 - ❖ SIMPLE_BUS_WAIT
 - ✦ The request is being served but not completed yet
 - ❖ SIMPLE_BUS_OK
 - ✦ The request is completed without errors
 - ❖ SIMPLE_BUS_ERROR
 - ✦ The request is finished but the transfer is not complete successfully
- ◆ Slave request status (read by the bus)
 - ❖ SIMPLE_BUS_WAIT
 - ✦ The slave issues a wait state
 - ❖ SIMPLE_BUS_OK
 - ✦ The transfer was successful
 - ❖ SIMPLE_BUS_ERROR
 - ✦ An error occurs during the transfer

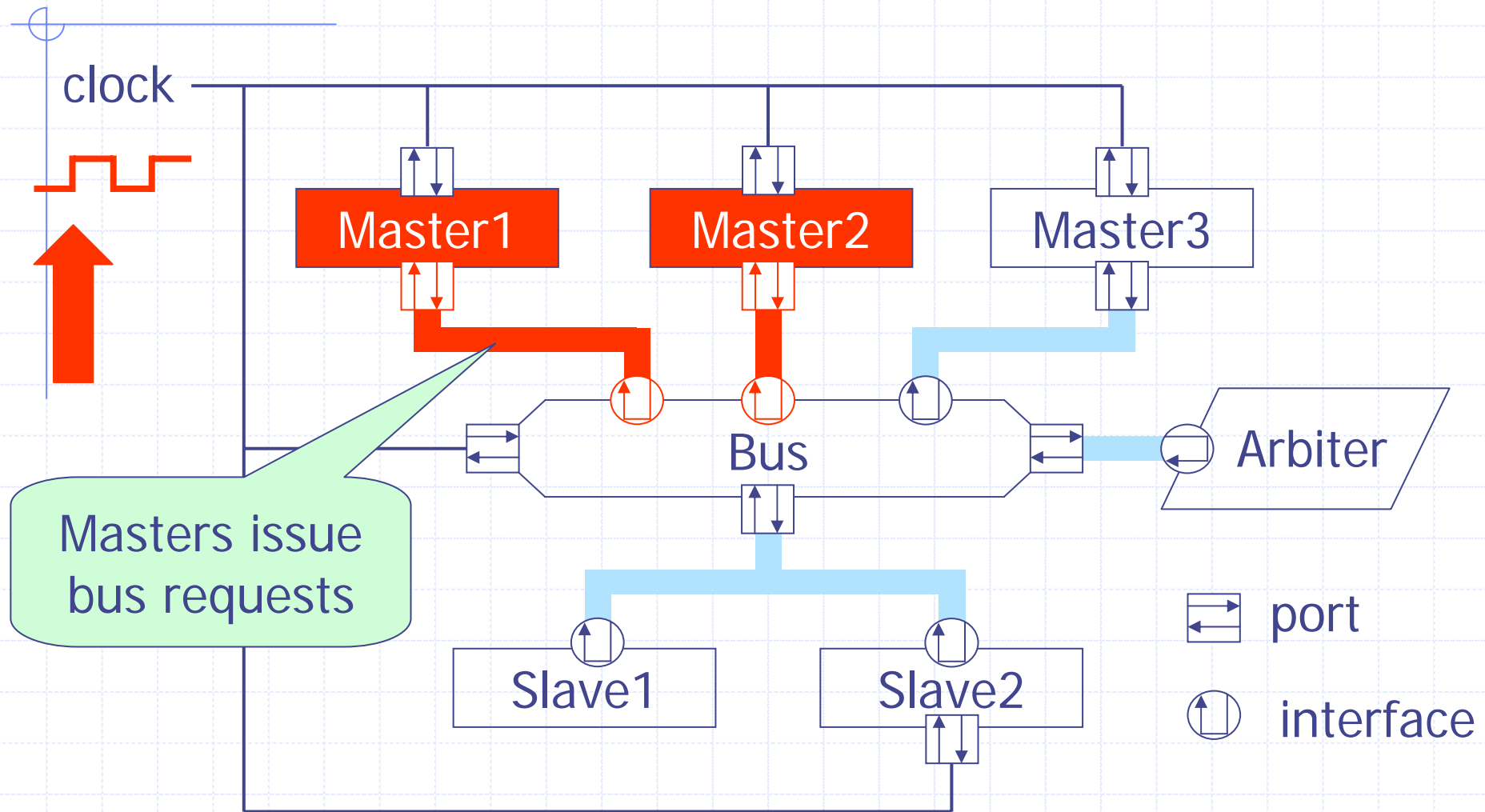


Concepts of Operations

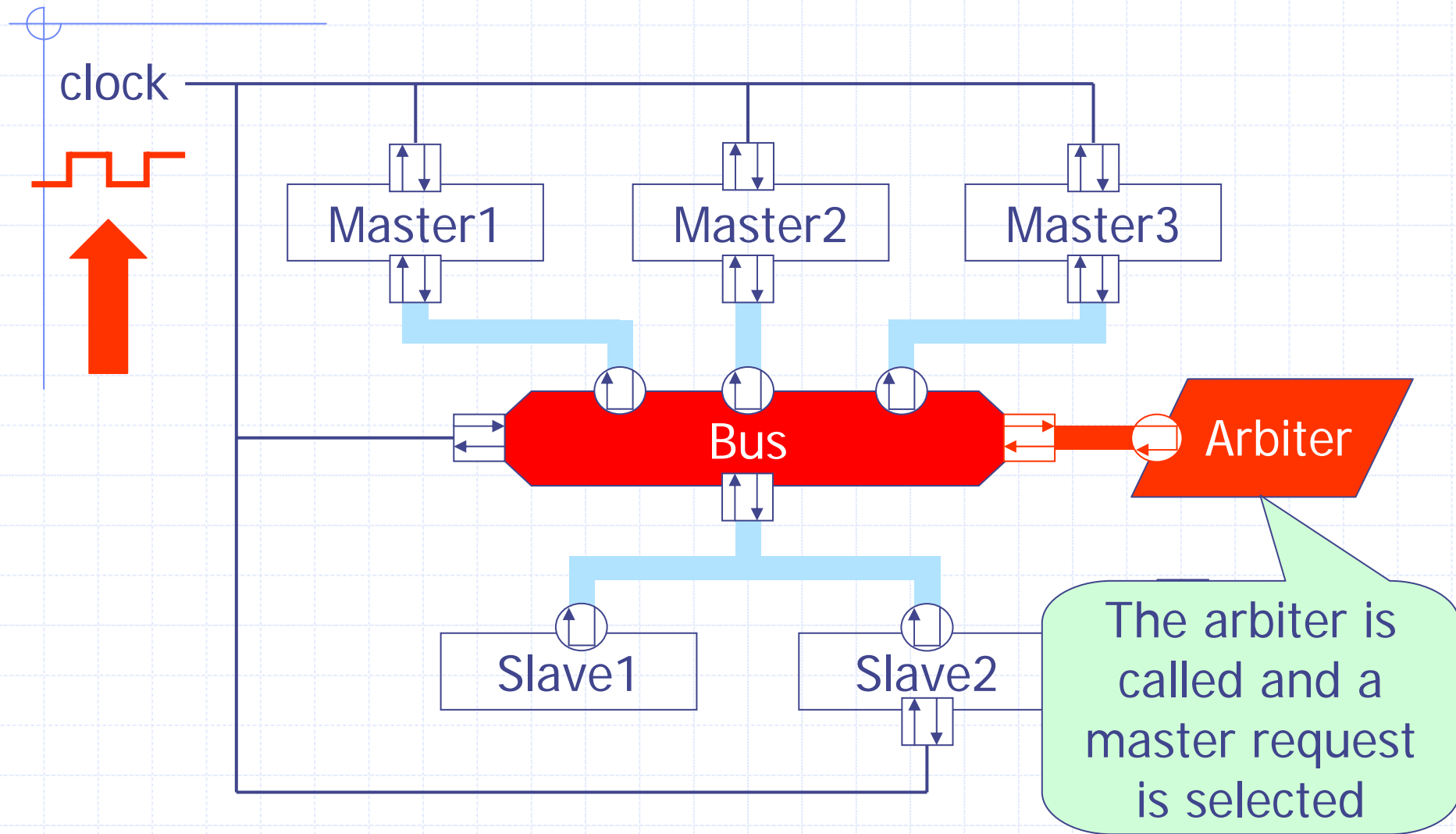
Overall Execution Scheme

- ◆ On the rising edge of the clock
 - ❖ Masters execute and may send requests to the bus
 - ❖ Bus maintains a set of outstanding requests including unfinished ones from past cycles
- ◆ On the falling edge of the clock
 - ❖ Bus calls arbiter to select a request for execution
 - ❖ Bus looks up the address of the request to determine the target slave
 - ❖ Bus invokes the read()/write() functions of the target slave
 - ❖ Functions return and indicate if the slave issues wait states
 - ✦ Bus will reissue the request on the next cycle upon receiving wait states
 - ❖ Bus updates the status of the original master once the slave completes the request

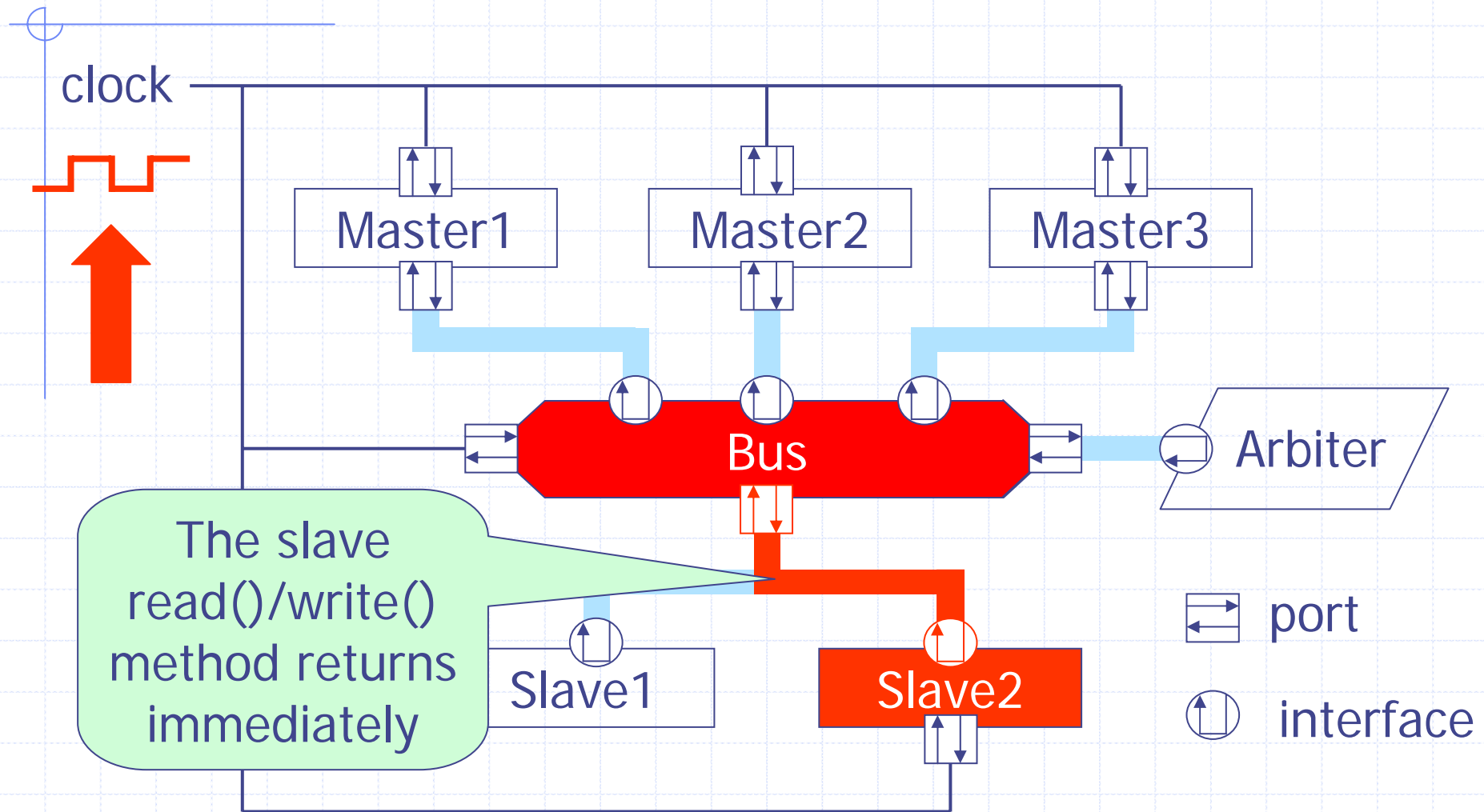
Overall Execution Scheme (c. 1) – Positive Edge



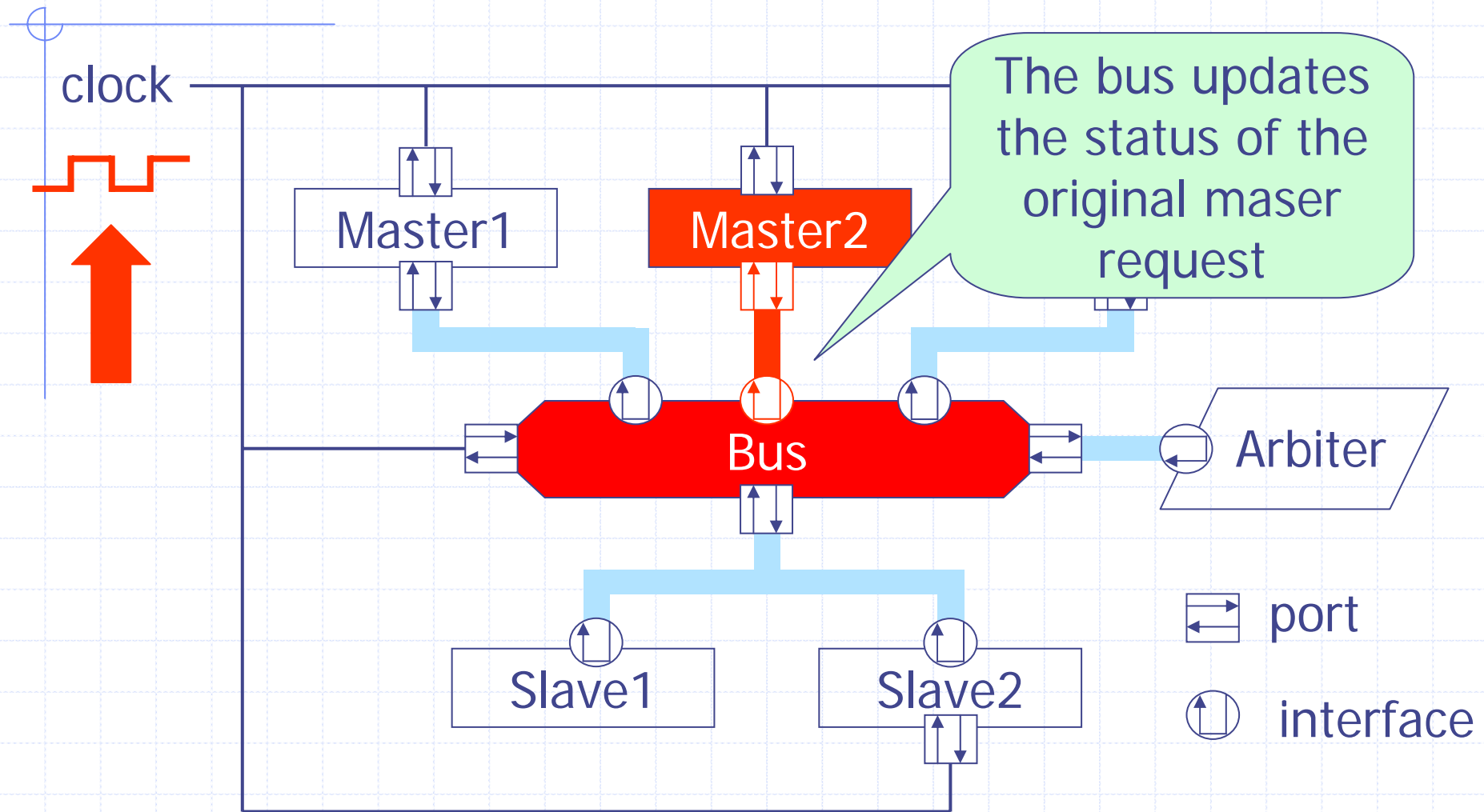
Overall Execution Scheme (c. 2) – Negative Edge



Overall Execution Scheme (c. 3) – Negative Edge



Overall Execution Scheme (c. 4) – Negative Edge



Two-Phase Synchronization

- ◆ Masters and slaves are active on the rising edge of the clock
- ◆ Bus and arbiters are active on the falling edge of the clock
- ◆ Two-phase synchronization
 - ❖ Communication between modules attached to the bus go through the bus
 - ❖ Communication is delayed by a clock cycle
 - ❖ On the rising edge of the clock, no state changes of the bus are visible
 - ❖ On the falling edge of the clock, the bus arbitrates the competing requests
- ◆ Request-update mechanism
 - ❖ Communications between processes go through the primitive channels
 - ❖ Communication is delayed by a delta-cycle
 - ❖ In the evaluation phase, no state changes of primitive channels are visible
 - ❖ In the update phase, primitive channels resolve competing requests

Two-Phase Synchronization (c. 1)

- ◆ Triggering the bus using the clock falling edge is just a technique
- ◆ Actual implementation may not use the falling edge of the clock
- ◆ Designs with the two-phase synchronization and deterministic arbitration rules are deterministic
 - ❖ The order of process execution will not affect the execution results



Implementation of Masters

Blocking Master

```
SC_MODULE(simple_bus_master_blocking)
{
    // ports
    sc_in_clk clock;
    sc_port<simple_bus_blocking_if> bus_port;

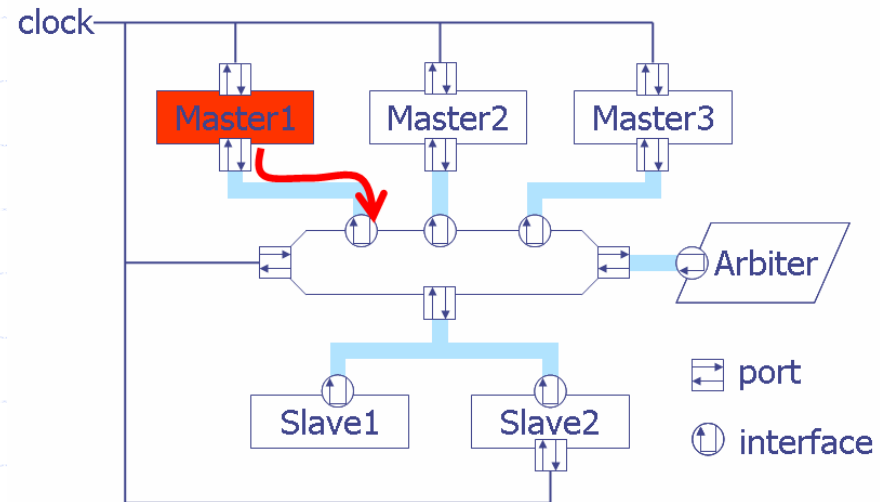
    SC_HAS_PROCESS(simple_bus_master_blocking);

    // constructor
    simple_bus_master_blocking(sc_module_name name_,
                              unsigned int unique_priority,
                              unsigned int address,
                              bool lock,
                              int timeout)
        : sc_module(name_)
        , m_unique_priority(unique_priority)
        , m_address(address)
        , m_lock(lock)
        , m_timeout(timeout)
    {
        // process declaration
        SC_THREAD(main_action);
        sensitive_pos << clock;

        // process
        void main_action();

private:
    unsigned int m_unique_priority;
    unsigned int m_address;
    bool m_lock;
    int m_timeout;
}; // end class simple_bus_master_blocking
```

Issue requests at the positive edge of clock



```
void simple_bus_master_blocking::main_action()
{
    // storage capacity/burst length in words
    const unsigned int mylength = 0x10;
    int mydata[mylength];
    unsigned int i;
    simple_bus_status status;

    while (true)
    {
        wait(); // ... for the next rising clock edge
        status = bus_port->burst_read(m_unique_priority, mydata,
                                     m_address, mylength, m_lock);

        for (i = 0; i < mylength; ++i)
        {
            mydata[i] += i;
            wait();
        }

        status = bus_port->burst_write(m_unique_priority, mydata,
                                     m_address, mylength, m_lock);

        wait(m_timeout, SC_NS);
    }
}
```

Blocking burst read

Wait for "mylength" cycles

Blocking burst write

Non-blocking Master

```

void simple_bus_master_non_blocking::main_action()
{
    int mydata;
    int cnt = 0;
    unsigned int addr = m_start_address;

    wait(); // ... for the next rising clock edge
    while (true)
    {
        bus_port->read(m_unique_priority, &mydata, addr, m_lock);
        while ((bus_port->get_status(m_unique_priority) != SIMPLE_BUS_OK) &&
               (bus_port->get_status(m_unique_priority) != SIMPLE_BUS_ERROR))
            wait();

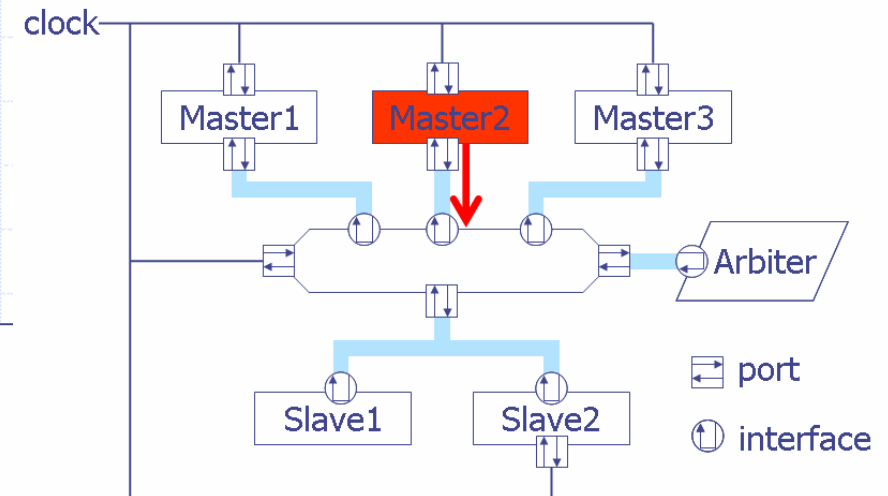
        mydata += cnt;
        cnt++;

        bus_port->write(m_unique_priority, &mydata, addr, m_lock);
        while ((bus_port->get_status(m_unique_priority) != SIMPLE_BUS_OK) &&
               (bus_port->get_status(m_unique_priority) != SIMPLE_BUS_ERROR))
            wait();

        wait(m_timeout, SC_NS);
        wait(); // ... for the next rising clock edge

        addr+=4; // next word (byte addressing)
        if (addr > (m_start_address+0x80)) {
            addr = m_start_address; cnt = 0;
        }
    }
}

```



Non-blocking read

Status polling

Non-blocking write

Status polling

Direct Master

```
SC_MODULE(simple_bus_master_direct)
{
    // ports
    sc_in_clk clock;
    sc_port<simple_bus_direct_if> bus_port;

    SC_HAS_PROCESS(simple_bus_master_direct);

    // constructor
    simple_bus_master_direct(sc_module_name name_,
                            unsigned int address,
                            int timeout,
                            bool verbose = true)
        : sc_module(name_)
        , m_address(address)
        , m_timeout(timeout)
        , m_verbose(verbose)
    {
        // process declaration
        SC_THREAD(main_action);

        // process
        void main_action();
    }

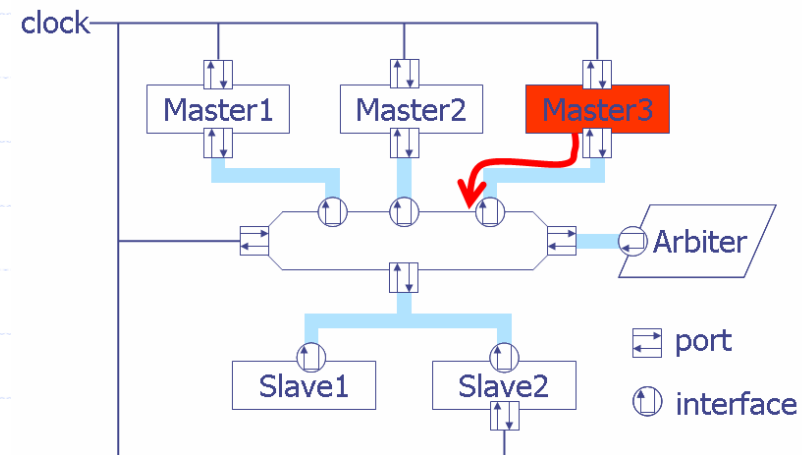
private:
    unsigned int m_address;
    int m_timeout;
    bool m_verbose;
}; // end class simple_bus_master_direct
```

Thread process with no static sensitivity

```
void simple_bus_master_direct::main_action()
{
    int mydata[4];
    while (true)
    {
        bus_port->direct_read(&mydata[0], m_address);
        bus_port->direct_read(&mydata[1], m_address+4);
        bus_port->direct_read(&mydata[2], m_address+8);
        bus_port->direct_read(&mydata[3], m_address+12);

        if (m_verbose)
            sb_fprintf(stdout, "%g %s : mem[%x:%x] = (%x, %x, %x, %x)\n",
                        sc_simulation_time(), name(), m_address, m_address+15,
                        mydata[0], mydata[1], mydata[2], mydata[3]);

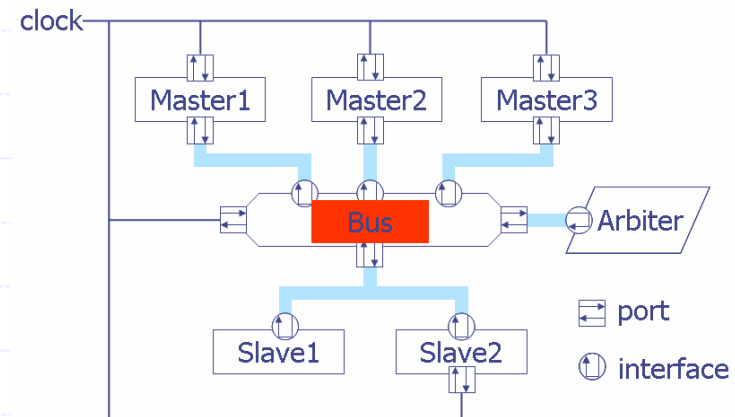
        wait(m_timeout, SC_NS);
    }
}
```





Implementation of Bus

Bus Implementation



```
class simple_bus
: public simple_bus_direct_if
, public simple_bus_non_blocking_if
, public simple_bus_blocking_if
, public sc_module
{
public:
// ports
sc_in_clk clock;
sc_port<simple_bus_arbiter_if> arbiter_port;
sc_port<simple_bus_slave_if, 0> slave_port;

SC_HAS_PROCESS(simple_bus);

// constructor
simple_bus(sc_module_name name_
, bool verbose = false)
: sc_module(name_)
, m_verbose(verbose)
, m_current_request(0)
{
// process declaration
SC_METHOD(main_action);
dont_initialize();
sensitive_neg << clock;
}

// process
void main_action();
}
```

Handle requests at the negative edge of clock

```
typedef sc_pvector<simple_bus_request *>
\ simple_bus_request_vec;
```

```
private:
void handle_request();
void end_of_elaboration();
simple_bus_slave_if * get_slave(unsigned int address);
simple_bus_request * get_request(unsigned int priority);
simple_bus_request * get_next_request();
void clear_locks();
```

Requests queued in the bus

```
private:
bool m_verbose;
simple_bus_request_vec m_requests;
simple_bus_request *m_current_request;
```

Request granted for execution

```
; // end class simple_bus
```

```
void simple_bus::main_action()
```

```
{
// m_current_request is cleared after the slave is done with a
// single data transfer. Burst requests require the arbiter to
// select the request again.
```

Request collection and arbitration

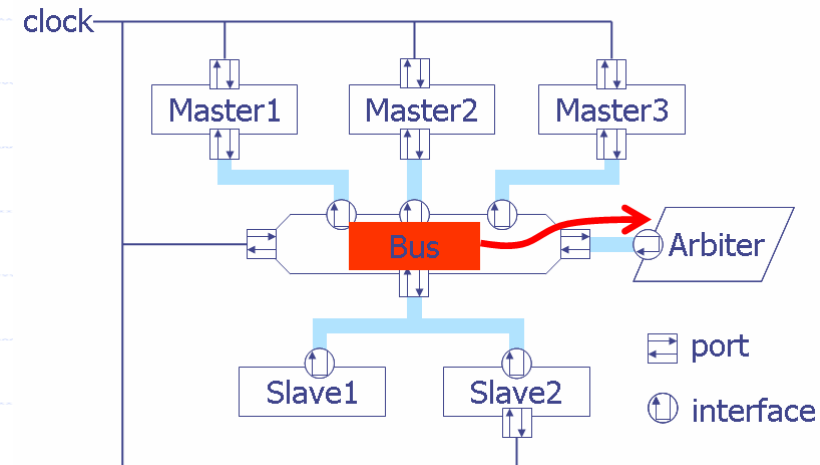
```
if (!m_current_request)
m_current_request = get_next_request();
```

```
if (m_current_request)
handle_request();
```

Request execution

```
if (!m_current_request)
clear_locks();
}
```

Request Collection



Collect bus requests in vector Q

```
simple_bus_request * simple_bus::get_next_request()
{
    // the slave is done with its action, m_current_request is
    // empty, so go over the bag of request-forms and compose
    // a set of likely requests. Pass it to the arbiter for the
    // final selection
    simple_bus_request_vec Q;
    for (int i = 0; i < m_requests.size(); ++i)
    {
        simple_bus_request *request = m_requests[i];
        if ((request->status == SIMPLE_BUS_REQUEST) ||
            (request->status == SIMPLE_BUS_WAIT))
        {
            if (m_verbose)
                sb_fprintf(stdout, "%g %s : request (%d) [%s]\n",
                           sc_simulation_time(), name(),
                           request->priority, simple_bus_status_str[request->status]);
            Q.push_back(request);
        }
    }
    if (Q.size() > 0)
        return arbiter_port->arbitrate(Q);
    return (simple_bus_request *)0;
}
```

Pass bus requests as a vector to arbiter and return the request to be executed

Request Arbitration Rules

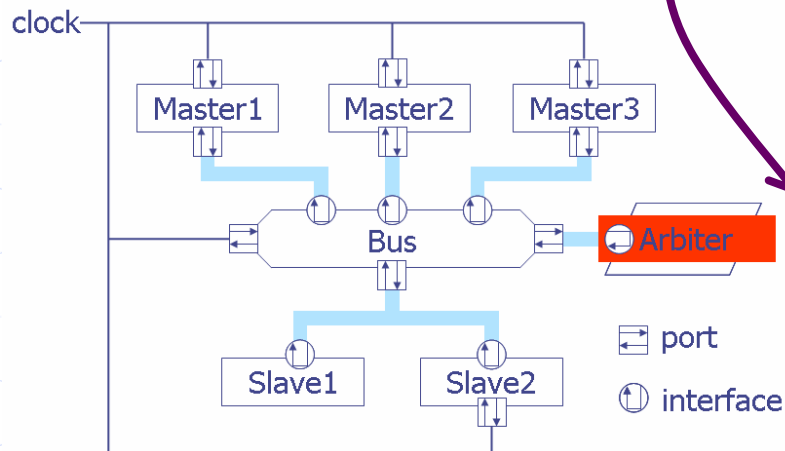
- ◆ If the request that was executed has its “lock” flag set, when the master issue the requests to the bus
 - ❖ If the request was a burst request and it is not yet completed, it is always selected
 - ❖ If the master that issued the request is issuing another request in the current cycle, then the master’s request is always selected

Request Arbitration

Highest Priority
Request that is being executed
and has been locked

Second Priority
Request that is set to lock
at previous call

Third Priority
Request of lower priority
number



```

simple_bus_request *
simple_bus_arbiter::arbitrate(const simple_bus_request_vec &requests)
{
    int i;
    // at least one request is here
    simple_bus_request *best_request = requests[0];

    // highest priority: status==SIMPLE_BUS_WAIT and lock is set:
    // locked burst-action
    for (i = 0; i < requests.size(); ++i)
    {
        simple_bus_request *request = requests[i];
        if ((request->status == SIMPLE_BUS_WAIT) &&
            (request->lock == SIMPLE_BUS_LOCK_SET))
            // cannot break-in a locked burst
            return request;
    }

    // second priority: lock is set at previous call,
    // i.e. SIMPLE_BUS_LOCK_GRANTED
    for (i = 0; i < requests.size(); ++i)
        if (requests[i]->lock == SIMPLE_BUS_LOCK_GRANTED)
            return requests[i];

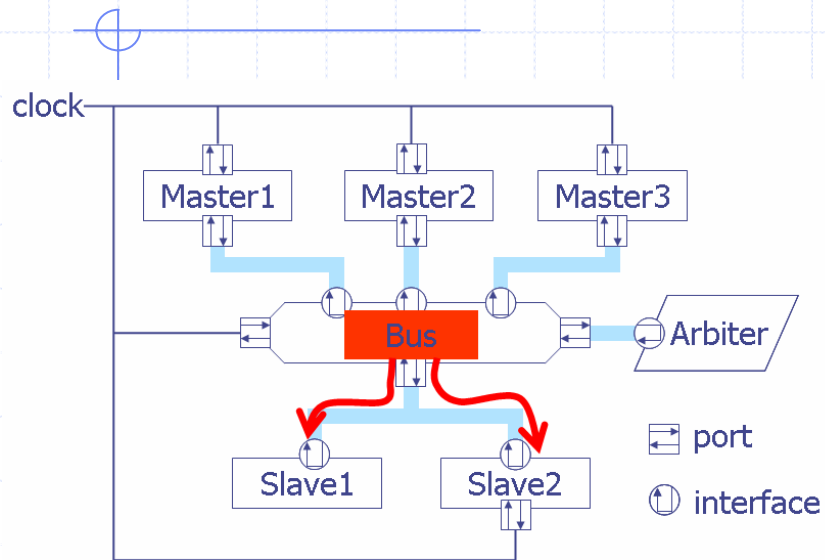
    // third priority: priority
    for (i = 1; i < requests.size(); ++i)
    {
        sc_assert(requests[i]->priority != best_request->priority);
        if (requests[i]->priority < best_request->priority)
            best_request = requests[i];
    }

    if (best_request->lock != SIMPLE_BUS_LOCK_NO)
        best_request->lock = SIMPLE_BUS_LOCK_GRANTED;

    return best_request;
}

```

Request Execution



```
simple_bus_slave_if *
simple_bus::get_slave(unsigned int address)
{
    for (int i = 0; i < slave_port.size(); ++i)
    {
        simple_bus_slave_if *slave = slave_port[i];
        if ((slave->start_address() <= address) &&
            (address <= slave->end_address()))
            return slave;
    }
    return (simple_bus_slave_if *)0;
}
```

Routing of the requests

```
void simple_bus::handle_request()
{
    m_current_request->status = SIMPLE_BUS_WAIT;
    simple_bus_slave_if *slave = get_slave(m_current_request->address);

    simple_bus_status slave_status = SIMPLE_BUS_OK;
    if (m_current_request->do_write)
        slave_status = slave->write(m_current_request->data,
                                    m_current_request->address);
    else
        slave_status = slave->read(m_current_request->data,
                                    m_current_request->address);

    switch(slave_status)
    {
        case SIMPLE_BUS_ERROR:
            m_current_request->status = SIMPLE_BUS_ERROR;
            m_current_request->transfer_done.notify();
            m_current_request = (simple_bus_request *)0;
            break;
        case SIMPLE_BUS_OK:
            m_current_request->address+=4; //next word (byte addressing)
            m_current_request->data++;
            if (m_current_request->address > m_current_request->end_address)
            {
                // burst-transfer (or single transfer)
                m_current_request->status = SIMPLE_BUS_OK;
                m_current_request->transfer_done.notify();
                m_current_request = (simple_bus_request *)0;
            }
            else
            {
                // more data to transfer, but the (atomic) slave transfer is done
                m_current_request = (simple_bus_request *)0;
            }
            break;
        case SIMPLE_BUS_WAIT:
            // the slave is still processing: no clearance of
            //the current request
            break;
        default:
            break;
    }
}
```

Master request status

Perform atomic slave read/write

Event notify

Clear request

Update master request

Event notify

Clear request

Clear request

Implementation of Blocking Interface

```

simple_bus_status simple_bus::burst_read(unsigned int unique_priority
    , int *data
    , unsigned int start_address
    , unsigned int length
    , bool lock)
{
    simple_bus_request *request = get_request(unique_priority);

    request->do_write      = false; // we are reading
    request->address       = start_address;
    request->end_address   = start_address + (length-1)*4;
    request->data          = data;

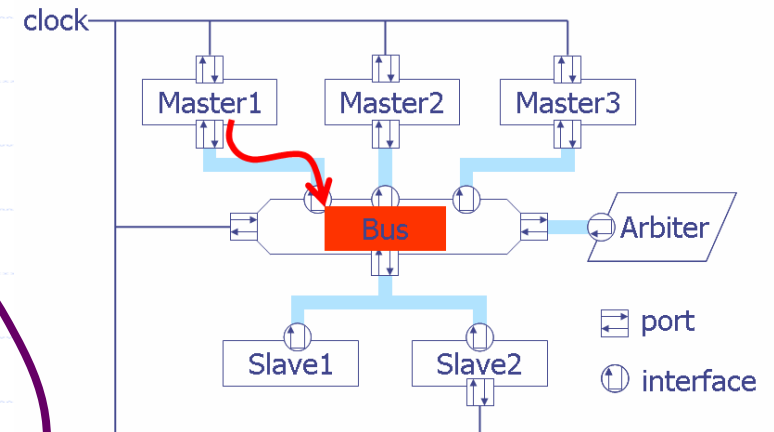
    if (lock)
        request->lock = (request->lock == SIMPLE_BUS_LOCK_SET) ?
            SIMPLE_BUS_LOCK_GRANTED : SIMPLE_BUS_LOCK_SET;

    request->status = SIMPLE_BUS_REQUEST;

    wait(request->transfer_done);
    wait(clock->posedge_event());
    return request->status;
}

```

Wait for the transaction to complete



```

simple_bus_request *
simple_bus::get_request(unsigned int priority)
{
    simple_bus_request *request = (simple_bus_request *)0;
    for (int i = 0; i < m_requests.size(); ++i)
    {
        request = m_requests[i];
        if ((request) &&
            (request->priority == priority))
            return request;
    }
    request = new simple_bus_request;
    request->priority = priority;
    m_requests.push_back(request);
    return request;
}

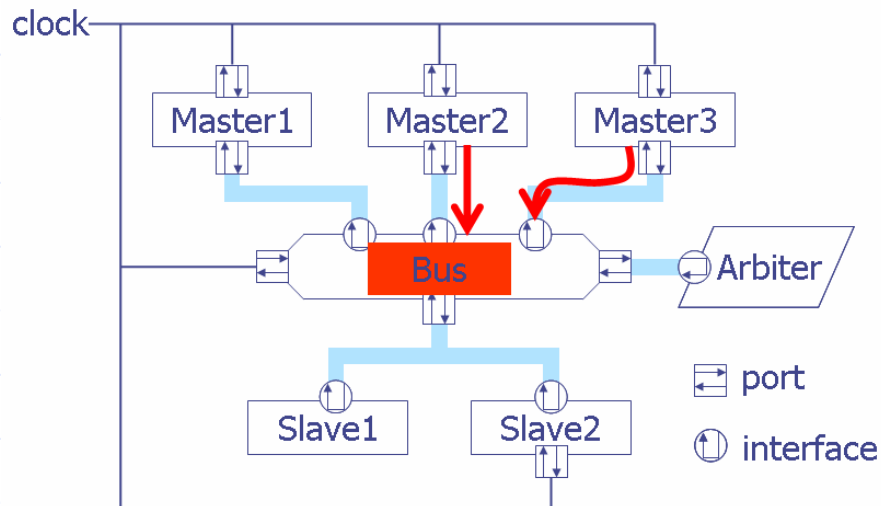
```

Implementation of Non-blocking, Direct Interfaces

```
bool simple_bus::direct_read(int *data, unsigned int address)
{
    simple_bus_slave_if *slave = get_slave(address);
    if (!slave) return false;
    return slave->direct_read(data, address);
}
```

Direct Read

Non-blocking Read



```
void simple_bus::read(unsigned int unique_priority
    , int *data
    , unsigned int address
    , bool lock)
{
    simple_bus_request *request = get_request(unique_priority);

    // abort when the request is still not finished
    sc_assert((request->status == SIMPLE_BUS_OK) ||
        (request->status == SIMPLE_BUS_ERROR));

    request->do_write      = false; // we are reading
    request->address       = address;
    request->end_address   = address;
    request->data          = data;

    if (lock)
        request->lock = (request->lock == SIMPLE_BUS_LOCK_SET) ?
            SIMPLE_BUS_LOCK_GRANTED : SIMPLE_BUS_LOCK_SET;

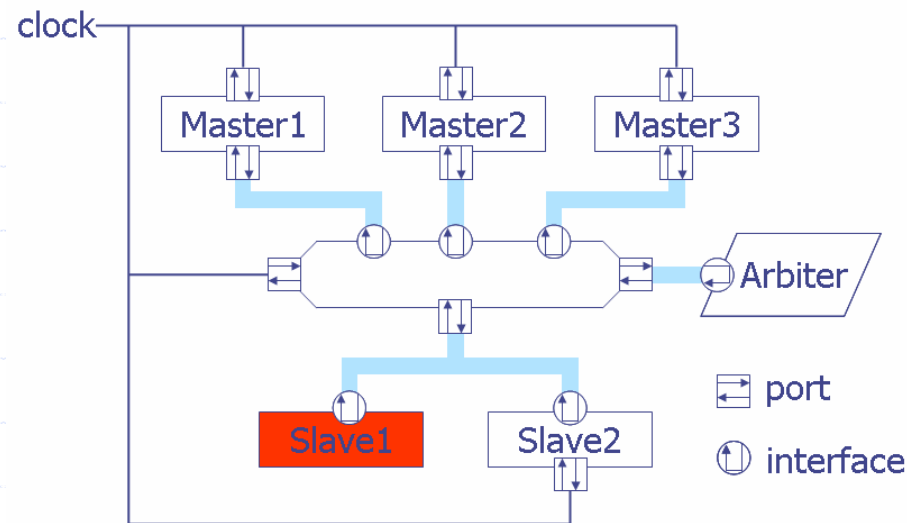
    request->status = SIMPLE_BUS_REQUEST;
}
```

No Wait Statements



Implementation of Slaves

Slave 1: Fast Memory



```

inline bool
simple_bus_fast_mem::direct_read(int *data, unsigned int address)
{
    return (read(data, address) == SIMPLE_BUS_OK);
}

inline simple_bus_status simple_bus_fast_mem::read(int *data
    , unsigned int address)
{
    *data = MEM[(address - m_start_address)/4];
    return SIMPLE_BUS_OK;
}

```

```

class simple_bus_fast_mem
: public simple_bus_slave_if
, public sc_module
{
public:
    // constructor
    simple_bus_fast_mem(sc_module_name name_
        , unsigned int start_address
        , unsigned int end_address)
        : sc_module(name_)
        , m_start_address(start_address)
        , m_end_address(end_address)
    {
        unsigned int size =
            (m_end_address - m_start_address + 1) / 4;
        MEM = new int [size];
        for (unsigned int i = 0; i < size; ++i)
            MEM[i] = 0;
    }

    // direct Slave Interface
    .....

    // Slave Interface
    .....

    unsigned int start_address() const;
    unsigned int end_address() const;

private:
    int * MEM;
    unsigned int m_start_address;
    unsigned int m_end_address;
}; // end class simple_bus_fast_mem

```

Slave 2: Slow Memory

Initiate the counters of wait states.
Bus will keep invoking the same
function until the return is
SIMPLE_BUS_OK.

```
inline simple_bus_status simple_bus_slow_mem::read(int *data
, unsigned int address)
{
    // accept a new call if m_wait_count < 0)
    if (m_wait_count < 0)
    {
        m_wait_count = m_nr_wait_states;
        return SIMPLE_BUS_WAIT;
    }
    if (m_wait_count == 0)
    {
        *data = MEM[(address - m_start_address)/4];
        return SIMPLE_BUS_OK;
    }
    return SIMPLE_BUS_WAIT;
}
```

```
inline void simple_bus_slow_mem::wait_loop()
{
    if (m_wait_count >= 0) m_wait_count--;
}

inline bool
simple_bus_slow_mem::direct_read(int *data, unsigned int address)
{
    *data = MEM[(address - m_start_address)/4];
    return true;
}
```

Works to be done are minimized in
the frequently activated method process

Do nothing without
read/write request

```
class simple_bus_slow_mem
: public simple_bus_slave_if
, public sc_module
{
public:
    // ports
    sc_in_clk clock;

    SC_HAS_PROCESS(simple_bus_slow_mem);

    // constructor
    simple_bus_slow_mem(sc_module_name name_
, unsigned int start_address
, unsigned int end_address
, unsigned int nr_wait_states)
: sc_module(name_)
, m_start_address(start_address)
, m_end_address(end_address)
, m_nr_wait_states(nr_wait_states)
, m_wait_count(-1)
{
    // process declaration
    SC_METHOD(wait_loop);
    dont_initialize();
    sensitive_pos << clock;
    .....
}

// process
void wait_loop();

// direct Slave Interface
.....
// Slave Interface
.....
unsigned int start_address() const;
unsigned int end_address() const;

private:
    int *MEM;
    unsigned int m_start_address;
    unsigned int m_end_address;
    unsigned int m_nr_wait_states;
    int m_wait_count;
}; // end class simple_bus_slow_mem
```

Method process



High-Performance Modeling Techniques

High-Performance Modeling Techniques

- ◆ Simple modules are modeled without any processes at all
 - ❖ Example: fast_mem and arbiter
- ◆ Blocks to be activated most frequently should use SC_METHOD
 - ❖ SC_METHOD consumes less memory and execute more quickly
- ◆ Frequently activated processes should do as little work as possible
 - ❖ Example: in slow_mem, there is a clocked SC_METHOD that simply decrements a counter to indicate when the wait states comes to completion

Comparisons between TLM and RTL

- ◆ RTL uses signals for communication; TLM employs transactions
 - ❖ Transactions are modeled by function calls
 - ❖ Both control and data are transferred along with function calls
 - ✦ There is no pin-accuracy
 - ✦ Data can be bundled and passed more efficiently
- ◆ Pointers to data are transferred between modules by transaction
 - ❖ Enable one module to very efficiently copy blocks of data to another
 - ❖ Example: the burst_read/burst_write transactions
- ◆ RTL uses low-level bit vectors; TLM uses high level C data-types
- ◆ RTL uses static sensitivity; TLM uses dynamic sensitivity
 - ❖ RTL modules execute on every cycle even if no work is being done
 - ❖ TLM modules enable execution when they have real work to perform
 - ✦ Processes are suspended until the bus requests complete

Common Questions

- ◆ What is the distinction between modules and hierarchical channels?
 - ❖ In an informal way
 - ✦ *Hierarchical channels*: implement interface functions and contain no ports
 - ✦ *Modules*: do not implement interface functions and contain ports
 - ❖ In reality
 - ✦ Hierarchical channels and modules are the same thing
- ◆ In simple_bus design
 - ❖ Blocks implementing transactions are designed to be channels that inherit from their transaction interface
 - ❖ Blocks that initiate transactions are designed to be modules that allow them to access the channels
 - ❖ The bus implements several interface functions and it also has ports to access the interface of the slaves and arbiter

Common Questions (c. 1)

- ◆ Why do slaves implement slave interface rather than having normal ports like other modules?
 - ❖ Eliminate the need for a process within the fast_mem and arbiter
 - ❖ Allow minimizing the amount of works in the process of slow_mem
- ◆ Why are multiple slave channels attached to the same port on the bus?
 - ❖ Do not want to fix the number of slaves
 - ❖ Allow binding as many slaves to the bus as wished during elaboration
 - ❖ Multi-port feature of SystemC
 - ✦ `sc_port<simple_bus_slave_if, 0> slave_port`
 - ✦ `slave_port.size()` returns the number of channels bounded to the port
 - ✦ `slave_port[N]` separates slave channels bounded to the port