Lecture 7 TLM API 1.0 in SystemC (Part II)

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

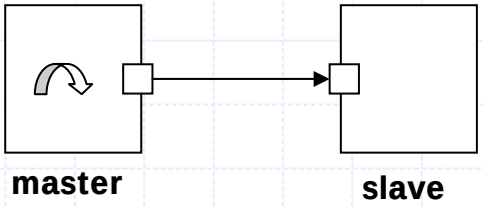
Reference

- Frank, Ghenassia, "*Transaction-Level Modeling with System C : TLM Concepts and Applications for Embedded Systems*", Springer, 2005. (ISBN: 0-387-26232-6)
- A. Rose, S. Swan, J. Pierce, and J.M Fernandez, "OSCI TLM Standard Whitepaper: Transaction Level Modeling in SystemC", http://www.systemc.org
- S. Swan, "A Tutorial Introduction to the SystemC TLM Standard", Cadence Design Systems, March 2006

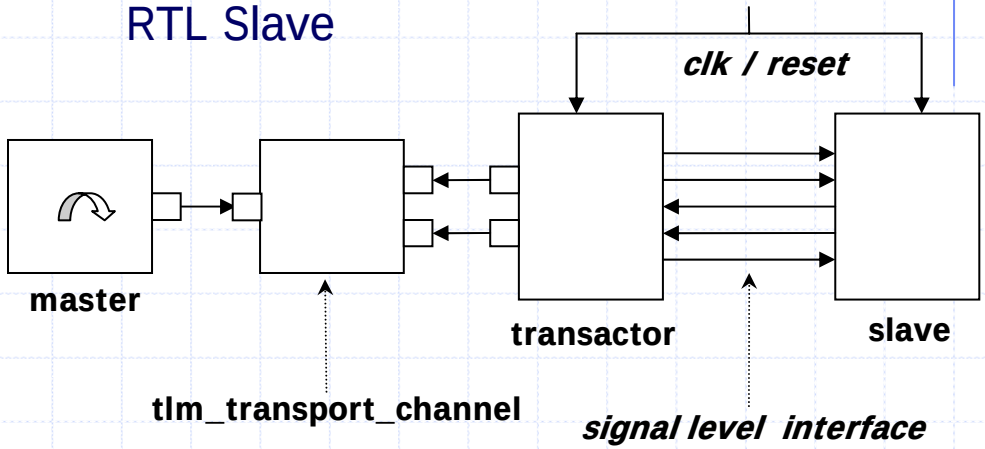
Examples of Using TLM API 1.0

Different Levels of Abstraction

- Programmer View (PV) Master/Slave

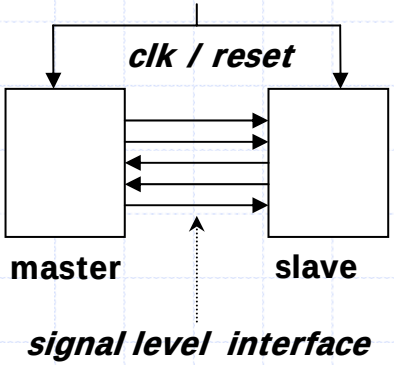


- PV Master + tlm transport channel + RTL Slave



- PV Master + tlm transport channel + Unidirectional Slave

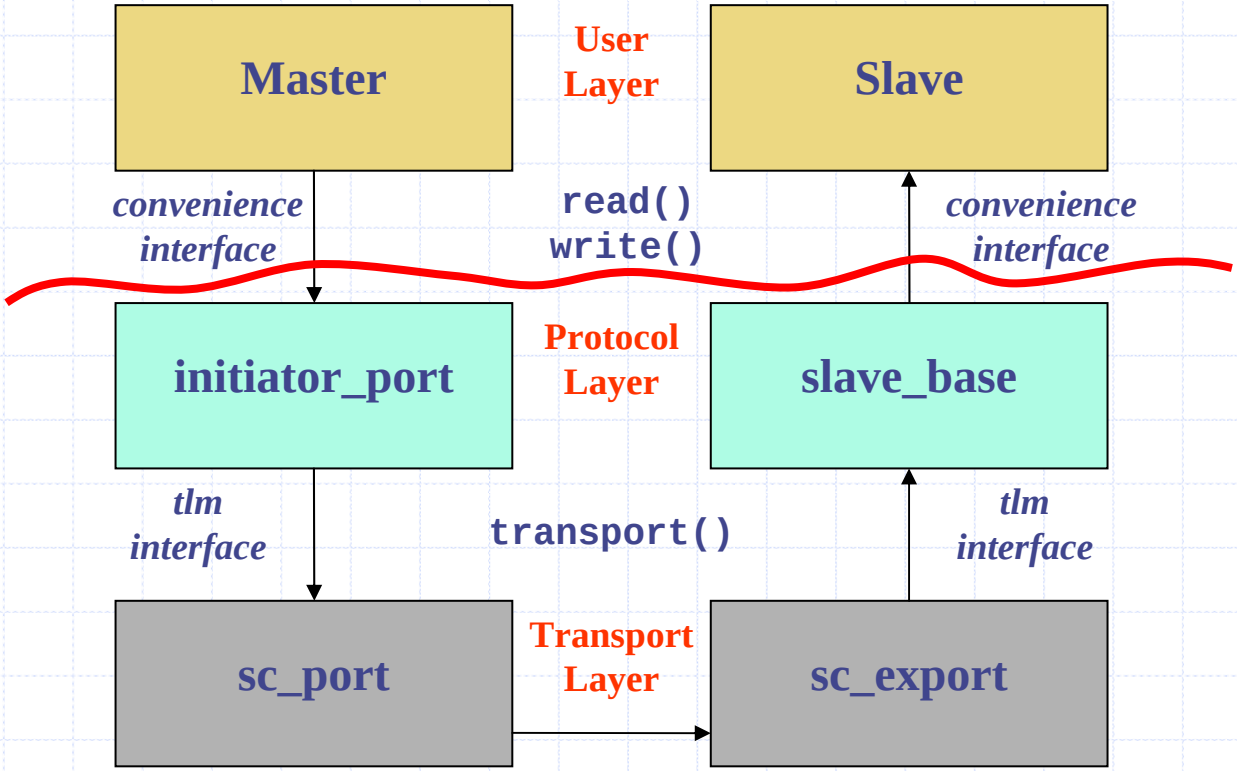
- RTL Master + RTL Slave



Protocol Stack Modeling Technique

- Allow different groups to exchange models easily

(1) Understand the application domain very well
(2) Not interested in the details of the TLM transport layer



(1) Not necessarily understand the application domain
(2) Understand the underlying protocols

Protocol Stack Modeling Technique (c. 1)

- Convenience interface
  - Offer methods that make sense to users of the protocol in question
    - Read, Write, Burst Read, Burst Write
  - Master
    - Use initiator ports that supply these interfaces
  - Slave
    - Define target modules which inherit from the these interfaces
- Protocol layer
  - Request and response classes
    - Encapsulate the protocol
  - An initiator port
    - Translate from convenience functions to RSP transport(const &REQ)
  - A slave base class
    - Implement RSP transport(const REQ &) in slave

Protocol Stack Modeling Technique (c. 2)

master

```
prot_initiator_port p;
p. read( a , d );
```

Slave

```
my_status read( Address a , Data &d );
my_status write( Address a , Data d );
```

USER LAYER

Port

Port

initiator_port

```
REQ req;
RSP rsp;
my_status read( Address a , Data &d ){
  req.a = a;
  req.type = read;
  rsp = transport( req );
  d = rsp.d;
  return rsp.status;
}
```

PROTOCOL LAYER

transaction_interface

```
virtual my_status read( Address a , Data &d ) = 0;
virtual my_status write( Address a , Data d ) = 0;
RSP transport( const REQ &req ){
   if( req.type == read )
      return read( req.addresss , rsp.data );
   else if( req.type == write )
      return write( req.address , req.data );
}
```

tlm_transaction

TRANSPORT LAYER

sc_port<REQ,RSP>

```
RSP transport( const REQ & );
```

tlm_transaction

sc_ export

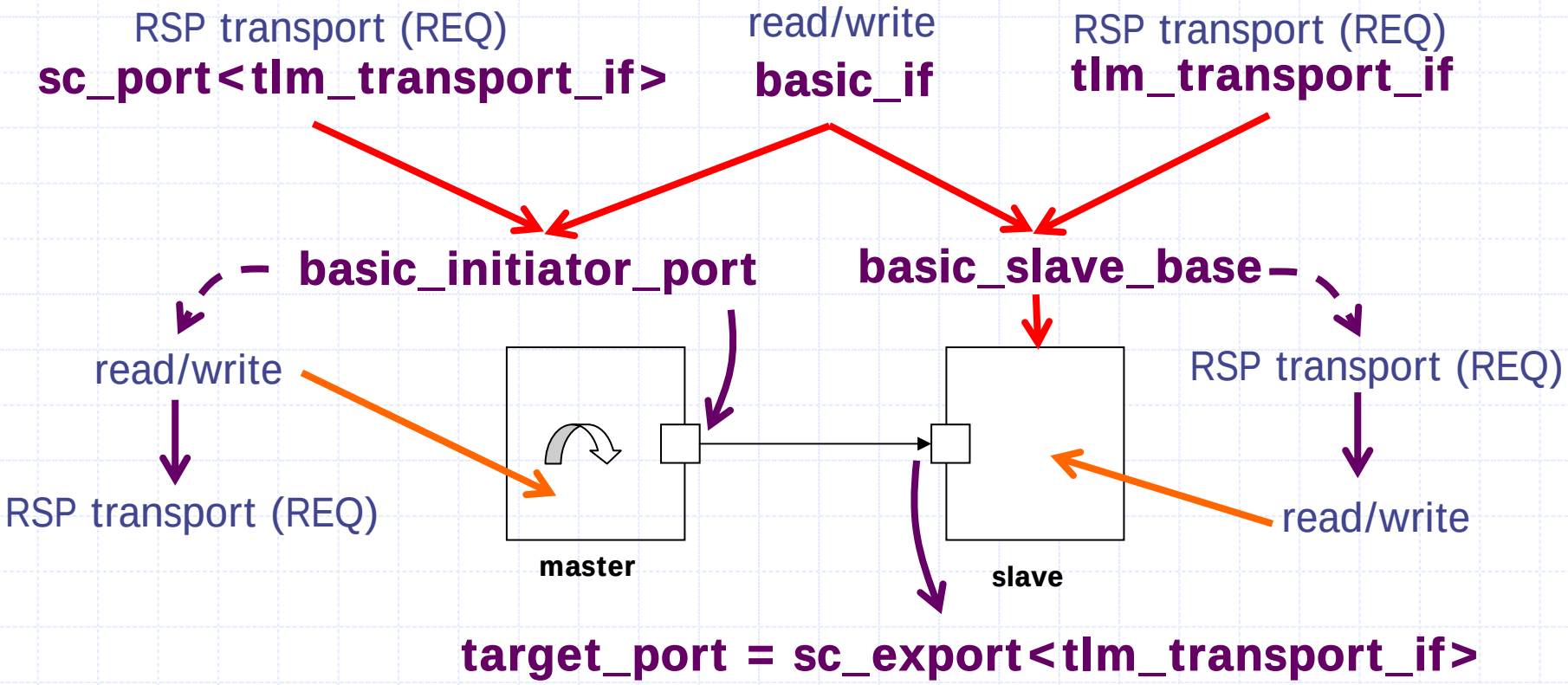
tlm_transport_if<REQ,RSP>

```
virtual RSP transport( const REQ & ) = 0;
```

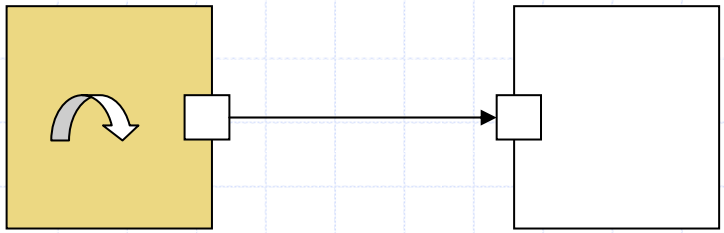
Programmer View (PV) Master and Slave

PV Master and Slave

- Use a single thread in the master to send a sequence of writes and reads to the slave



Master




```
int sc_main( int argc , char **argv ){
   master m("master");
   mem_slave s("slave");
   m.initiator_port( s.target_port );
}
```

```
class master : public sc_module{
public:
 master( sc_module_name module_name );
 SC_HAS_PROCESS( master );
 basic_initiator_port<ADDRESS_TYPE,DATA_TYPE>
                          initiator_port;
private:
  void run();
};
```

```
master::master( sc_module_name module_name )  :
   sc_module( module_name ) ,
   initiator_port("iport")
   {  SC_THREAD( run ); }

void master::run(){
  DATA_TYPE d;

  for( ADDRESS_TYPE a = 0; a < 20; a++ )
      initiator_port.write( a , a + 50 );

  for( ADDRESS_TYPE a = 0; a < 20; a++ )
      initiator_port.read( a , d );
}
```

basic_initiator_port

- Translate from convenience layer to transport layer
  - sc_port<tlm_transport_if<REQ, RSP> >
  - basic_if<ADDRESS, DATA>

```
template< typename ADDRESS , typename DATA>
class basic_request{
public:
  basic_request_type type;
  ADDRESS &get_address() { return a; }
  ADDRESS a;
  DATA d;
};
```

```
template< typename DATA >
class basic_response{
public:
  basic_response() : status( ERROR ) {}
  basic_request_type type;
  basic_status status;
  DATA d;
};
```

```
template< typename ADDRESS , typename DATA >
class basic_if
{
public:
virtual basic_status write( const ADDRESS &a , const DATA &d ) = 0;
virtual basic_status read( const ADDRESS &a , DATA &d ) = 0;
}
```

Convenience Interface

```
virtual basic_status write( const ADDRESS &a , const DATA &d ) {
  basic_request<ADDRESS,DATA> req;
  basic_response<DATA> rsp;
  req.type = WRITE;
  req.a = a;
  req.d = d;
  rsp = (*this)->transport( req );
 return rsp.status;
}
```

Convenience Interface to RSP transport (REQ)

basic_slave_base

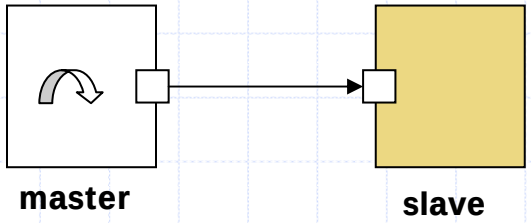
- Translate back from transport layer to convenience layer
  - tlm_transport_if<REQ, RSP>
  - basic_if<ADDRESS, DATA>

```
basic_response<DATA> transport( const basic_request<ADDRESS,DATA> &request ) {
    basic_response<DATA> response;
    switch( request.type ) {
        case READ :
            response.status = read( request.a , response.d );
            break;
        case WRITE:
            response.status = write( request.a , request.d );
            break;
        default :
            response.status = ERROR;
            break;
    }
    return response;
}
```

RSP transport (REQ)
to
Convenience Interface

Slave

- Directly implement the read/write functions called from transport interface function



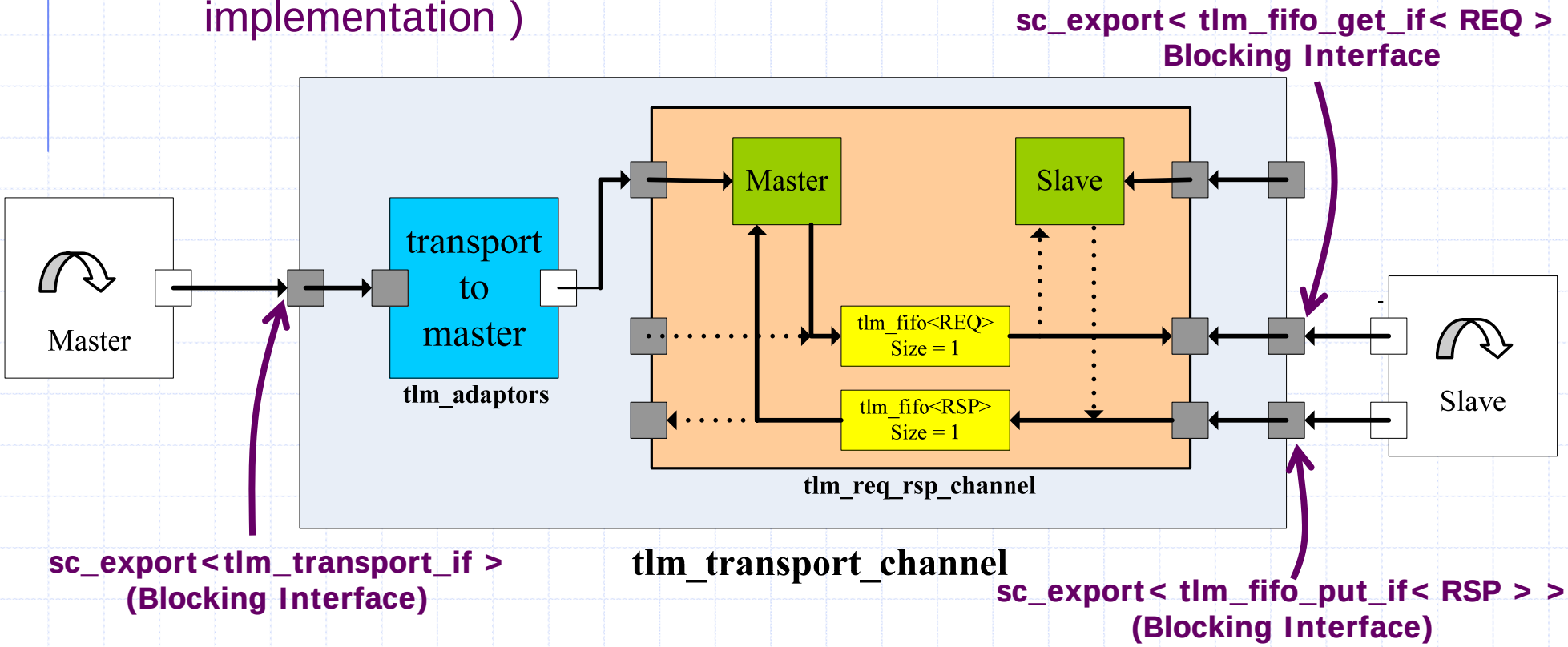

```
class mem_slave :
public sc_module ,
public virtual basic_slave_base< ADDRESS_TYPE , DATA_TYPE >
{
public:
mem_slave( sc_module_name module_name , int k = 10 );
sc_export< if_type > target_port;
basic_status write( const ADDRESS_TYPE & , const DATA_TYPE &);
basic_status read( const ADDRESS_TYPE & , DATA_TYPE & );
~mem_slave();
private:
  ADDRESS_TYPE *memory;
};
```

```
mem_slave::mem_slave( sc_module_name module_name ,
int k ) :  sc_module( module_name ) , target_port("iport")
{
  target_port( *this );
  memory = new ADDRESS_TYPE[ k * 1024 ];
}

basic_status mem_slave::write( const ADDRESS_TYPE
&a , const DATA_TYPE &d )
{
   memory[a] = d;
   return basic_protocol::SUCCESS;
}
```

PV Master + Tlm_Transport_Channel + Unidirectional Slave

PV Master and Unidirectional Slave

- Connect a master to a slave using tlm_transport_channel
  - PV Master uses bidirectional transport interface
  - Slave employs unidirectional interface (closer to the final implementation)



Tlm_Transport_to_Master (Adapter)

- Translate from transport interface to master interface
  - tlm_transport_if< REQ , RSP >, sc_module

```
class tlm_transport_to_master :
public sc_module ,
public virtual tlm_transport_if< REQ , RSP >
{
public:
  sc_export< tlm_transport_if< REQ , RSP > > target_export;
  sc_port< tlm_master_if< REQ , RSP > > master_port;

  tlm_transport_to_master( sc_module_name nm ) : sc_module( nm )
  {    target_export( *this );  }

  RSP transport( const REQ &req ) {
    mutex.lock();
    master_port->put( req );
    rsp = master_port->get();
    mutex.unlock();
    return rsp;
 }
private:
   sc_mutex mutex;
   RSP rsp;
};
```

Slave

- Model separately the request and response phases
  - Closer to the final implementation using 1 additional thread

```
class mem_slave :  public sc_module
{
 public:
   mem_slave( sc_module_name module_name , int k = 10 );
   SC_HAS_PROCESS( mem_slave );
   sc_port<tlm_blocking_get_if<REQ> > in_port;
   sc_port<tlm_blocking_put_if<RSP> > out_port;
   ~mem_slave();
private:
   void run();
   ADDRESS_TYPE *memory;
};
```

```
mem_slave::mem_slave( sc_module_name module_name ,
int k ) :  sc_module( module_name ) ,
           in_port("in_port") ,
           out_port("out_port")
{
    SC_THREAD( run );
    memory = new ADDRESS_TYPE[k * 1024];
}
```

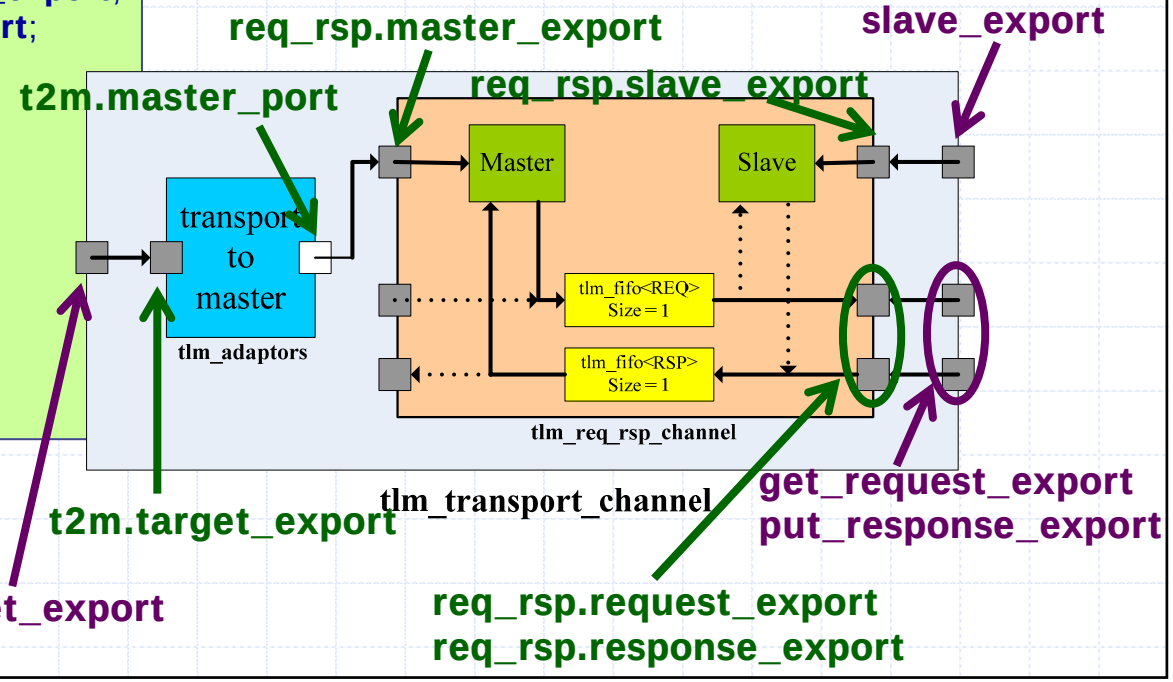
```
void mem_slave::run(){
basic_request<ADDRESS_TYPE,DATA_TYPE> request;
basic_response<DATA_TYPE> response;
for(;;)  {
   request = in_port->get();  //blocking tlm_fifo get
   response.type = request.type;
        switch( request.type ){
            case basic_protocol::READ :
                    response.d = memory[request.a];
                    response.status = basic_protocol::SUCCESS;
                    break;
            case basic_protocol::WRITE:
                    memory[request.a] = request.d;
                    response.status = basic_protocol::SUCCESS;
                    break;
            default:
                    response.status = basic_protocol::SUCCESS;
                    break;
        }
    out_port->put( response );  //blocking tlm_fifo put
}
}
```

Tlm_Transport_Channel

- Encapsulate tlm_transport_to_master and tlm_req_rsp_channel

```
template < typename REQ , typename RSP >
class tlm_transport_channel : public sc_module{
public:
// master transport interface
sc_export< tlm_transport_if< REQ , RSP > > target_export;
// slave interfaces
sc_export< tlm_fifo_get_if< REQ > > get_request_export;
sc_export< tlm_fifo_put_if< RSP > > put_response_export;
sc_export< tlm_slave_if< REQ , RSP > > slave_export;

tlm_transport_channel( sc_module_name nm ) :
sc_module( nm ) ,
target_export("target_export") ,
req_rsp( "req_rsp" , 1 , 1 ) ,
t2m("tsm" )  {    do_binding();  }

tlm_req_rsp_channel< REQ , RSP > req_rsp;
tlm_transport_to_master< REQ , RSP > t2m;
};
```

```
void do_binding() {
target_export( t2m.target_export );
t2m.master_port( req_rsp.master_export );
get_request_export( req_rsp.get_request_export );
put_response_export( req_rsp.put_response_export );
slave_export( req_rsp.slave_export );
}
```



Tlm_Req_Rsp_Channel

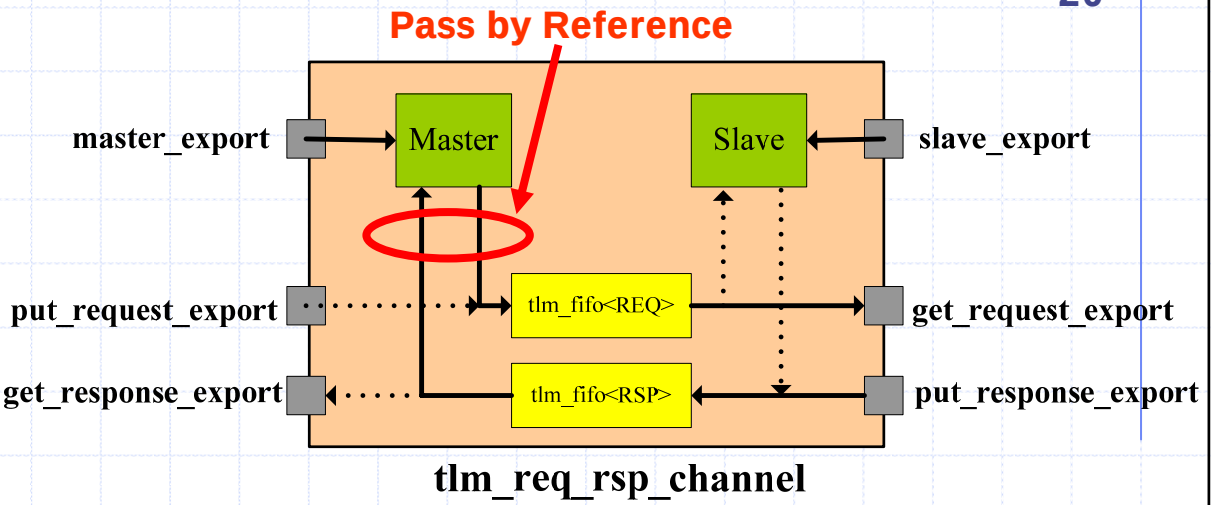
- Encapsulate 2 tlm_fifo of size 1 and tlm_master/slave_imp

```
template < typename REQ , typename RSP >
class tlm_req_rsp_channel : public sc_module{
public:
// uni-directional slave interface
sc_export< tlm_fifo_get_if< REQ > > get_request_export;
sc_export< tlm_fifo_put_if< RSP > > put_response_export;

// uni-directional master interface
sc_export< tlm_fifo_put_if< REQ > > put_request_export;
sc_export< tlm_fifo_get_if< RSP > > get_response_export;

// master / slave interfaces
sc_export< tlm_master_if< REQ , RSP > > master_export;
sc_export< tlm_slave_if< REQ , RSP > > slave_export;

.....
protected:
tlm_master_imp< REQ , RSP > master;
tlm_slave_imp< REQ , RSP > slave;
tlm_fifo<REQ> request_fifo;
tlm_fifo<RSP> response_fifo;
```

```
//Constructor of tlm_req_rsp_channel
tlm_req_rsp_channel( sc_module_name module_name,
int req_size = 1 , int rsp_size = 1 ) :
sc_module( module_name  ) ,
master( request_fifo , response_fifo ) ,
slave( request_fifo , response_fifo ) ,
request_fifo( req_size ) ,
response_fifo( rsp_size )
 {   bind_exports();  }
```

```
void bind_exports() {
put_request_export( request_fifo );
get_request_export( request_fifo );
put_response_export( response_fifo );
get_response_export( response_fifo );
master_export( master );
slave_export( slave );
}
```

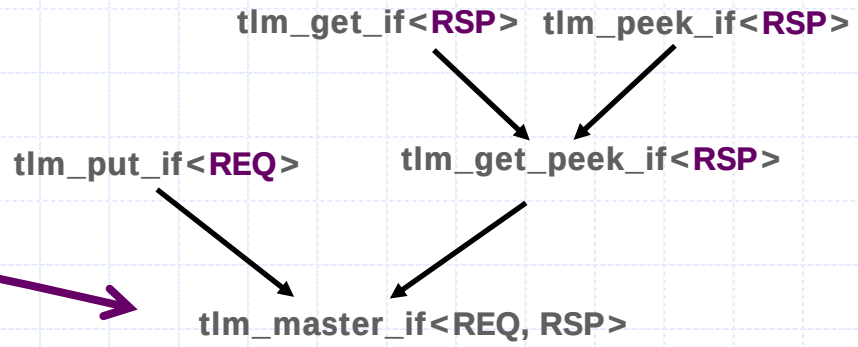
Tlm_Req_Rsp_Channel




```
template < typename REQ , typename RSP >
class tlm_req_rsp_channel : public sc_module{
public:
// uni-directional slave interface
sc_export< tlm_fifo_get_if< REQ > > get_request_export;
sc_export< tlm_fifo_put_if< RSP > > put_response_export;

// uni-directional master interface
sc_export< tlm_fifo_put_if< REQ > > put_request_export;
sc_export< tlm_fifo_get_if< RSP > > get_response_export;

// master / slave interfaces
sc_export< tlm_master_if< REQ , RSP > > master_export;
sc_export< tlm_slave_if< REQ , RSP > > slave_export;

.....
protected:
tlm_master_imp< REQ , RSP > master;
tlm_slave_imp< REQ , RSP > slave;
tlm_fifo<REQ> request_fifo;
tlm_fifo<RSP> response_fifo;
```

Encapsulate interfaces of tlm_fifo

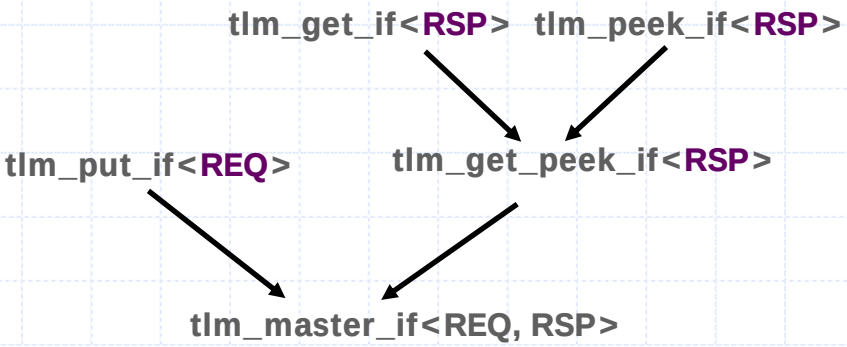
```
void bind_exports() {
put_request_export( request_fifo );
get_request_export( request_fifo );
put_response_export( response_fifo );
get_response_export( response_fifo );
master_export( master );
slave_export( slave );
}
```



tlm_master_imp

- Implementation of master channel

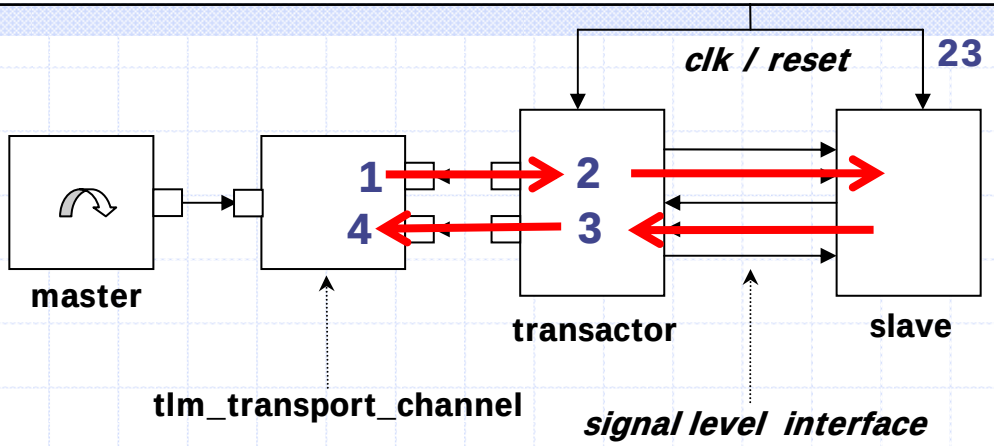
```
template < typename REQ , typename RSP >
class tlm_master_imp :
private tlm_put_get_imp< REQ , RSP > ,
public virtual tlm_master_if< REQ , RSP >
{
public:
tlm_master_imp( tlm_fifo<REQ> &req ,
tlm_fifo<RSP> &rsp ) :
tlm_put_get_imp<REQ,RSP>( req , rsp ) {}
};
```




```
template < typename PUT_DATA , typename GET_DATA >
class tlm_put_get_imp :
private virtual tlm_put_if< PUT_DATA > ,
private virtual tlm_get_peek_if< GET_DATA >
{
public:
tlm_put_get_imp( tlm_fifo<PUT_DATA> &p ,tlm_fifo<GET_DATA> &g ) :
put_fifo( p ) , get_fifo( g ) {}
// put interface
void put( const PUT_DATA &t ) { put_fifo.put( t ); }
bool nb_put( const PUT_DATA &t ) { return put_fifo.nb_put( t ); }
bool nb_can_put( tlm_tag<PUT_DATA> *t = 0 ) const
                        {    return put_fifo.nb_can_put( t );  }
const sc_event &ok_to_put( tlm_tag<PUT_DATA> *t = 0 ) const
                        {    return put_fifo.ok_to_put( t );  }
// get interface
.....
// peek interface
.....
private:
tlm_fifo<PUT_DATA> &put_fifo; //put_fifo reference
tlm_fifo<GET_DATA> &get_fifo; //get_fifo reference
};
```

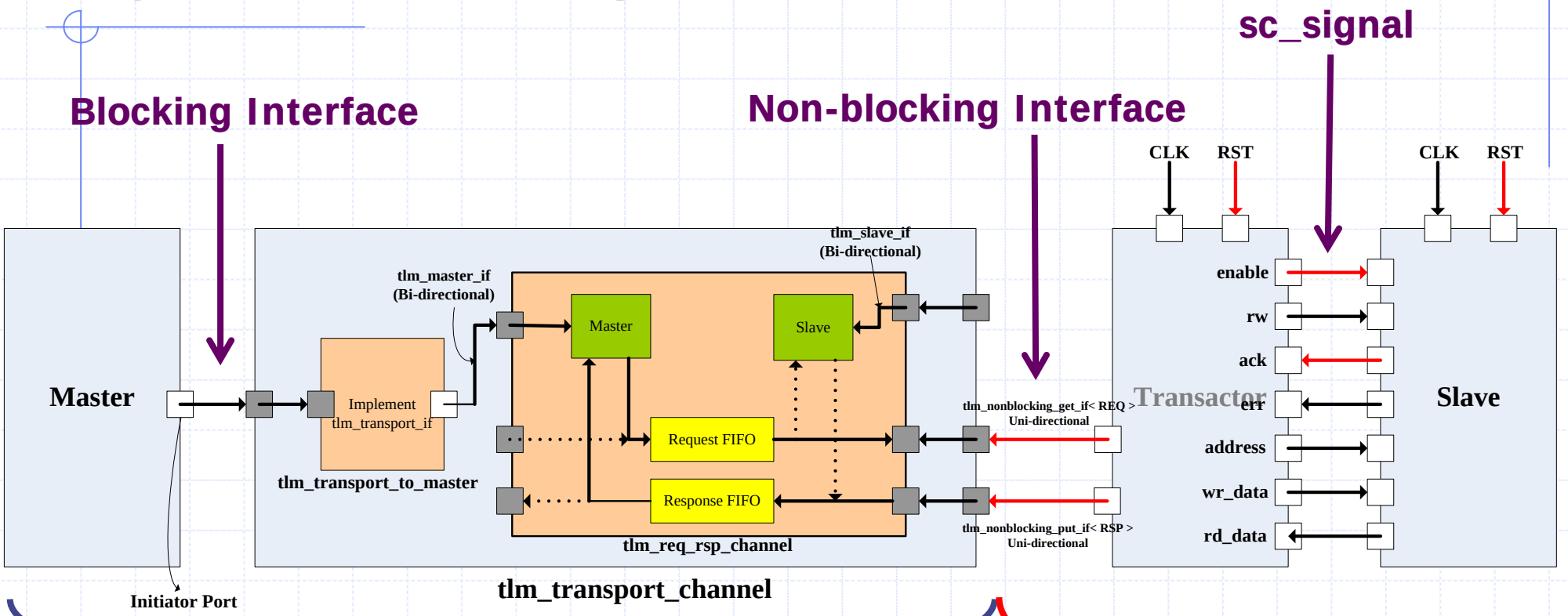
PV Master + Tlm_Transport_Channel + RTL Slave

System Architecture

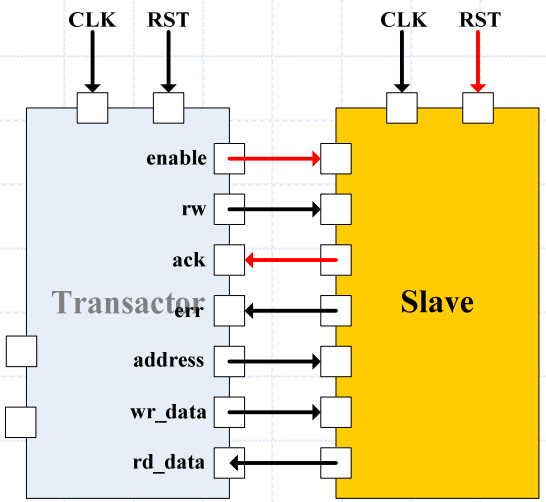


- Goal
  - Refine the slave further to a RTL implementation
- System components
  - Master – High Level Software
    - Send a request to tlm_transport_channel
    - Wait until a response is available in tlm_transport_channel
  - Transactor – key component
    1. Get an abstract request from the request fifo in transport_channel
    2. Enable the slave and send the request out over the rtl level bus
    3. Wait until it sees a “ack” response on the rtl bus
    4. Disable the slave and put the abstract response in the response fifo
  - Slave – Low Level RTL Implementation
    - Interact with transactor through sc_signal

System Block Diagram



Slave – RTL Implementation



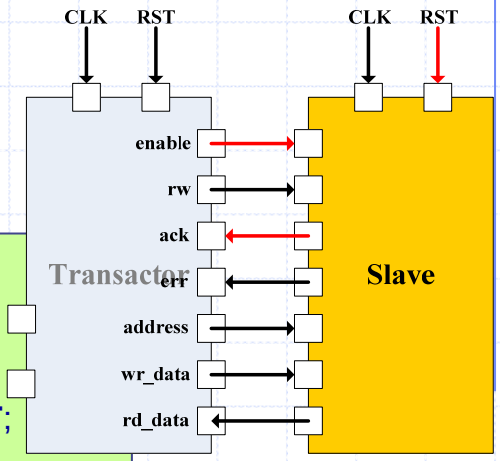

```
class slave : public sc_module
{
public:
slave( sc_module_name module_name );
SC_HAS_PROCESS( slave );

sc_in<bool> clk;
sc_in<bool> rst;
sc_in<bool> enable;
sc_in<bool> rw;
sc_out<bool>  ack;
sc_out<bool>  err;
sc_in< sc_uint< 8 > > address;
sc_in< sc_uint < 8 > >  wr_data;
sc_out< sc_uint < 8 > > rd_data;

private:
enum state {RESET ,    READY};
state m_state;
sc_signal<int> m_count;  //support request_update
void run();
sc_uint<8> memory[memory_size];
};
```

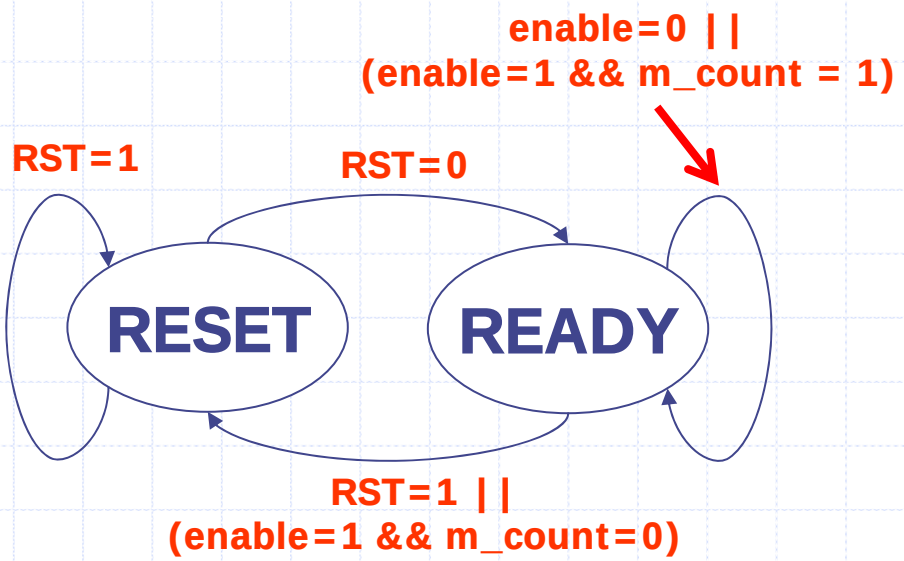
```
slave::slave( sc_module_name module_name ) :
sc_module( module_name ) ,
clk("clk") ,
rst("rst") ,
enable("en") ,
address("address") ,
rw("rw") ,
wr_data("wr_data" ) ,
rd_data("rd_data" ) ,
ack("ack") ,
err("err" )
{
 SC_METHOD( run );
 sensitive << clk.pos();
 dont_initialize();
}
```

Slave – RTL Implementation (c. 1)

```
switch( m_state ) {                    Finite State Machine
case RESET :
        ack = false;
        m_state = READY;
        m_count = ready_count; //ready_count = 1
        break;
case READY :
        if( enable ) {
             //Take effect in next delta cycle
             m_count = m_count - 1;
             if( m_count == 0 ) {
                  ack = true;
                  m_state = RESET;

                  if( address.read() >= memory_size ) err = true;
                  else                                err = false;

                  if( rw )
                       rd_data = memory[address.read()];
                  else
                      memory[address.read()] = wr_data.read();
             }else
                 ack = false;
        } else
             ack = false;//No change in system state
        break;
default:
       .....................................
```

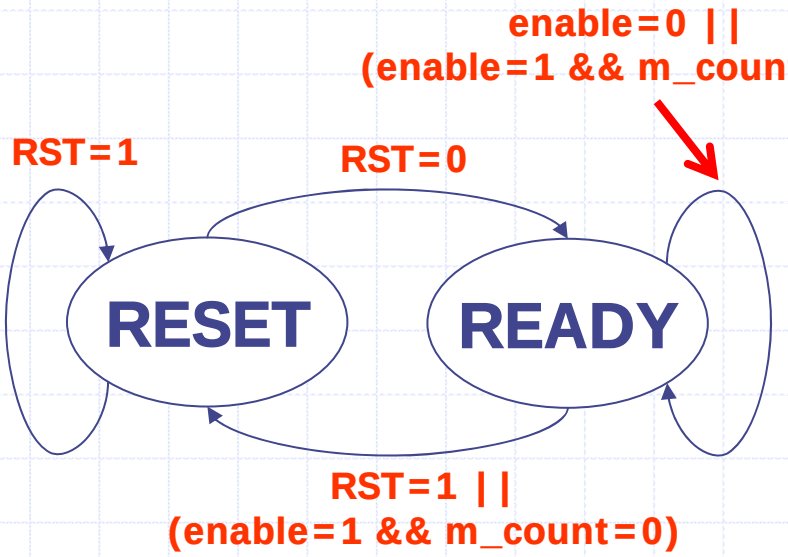
```
void run()
{
    if(rst )
    {
        m_state = RESET;
        ack = false;
    }else{
        Finite State Machine
    }
}
```

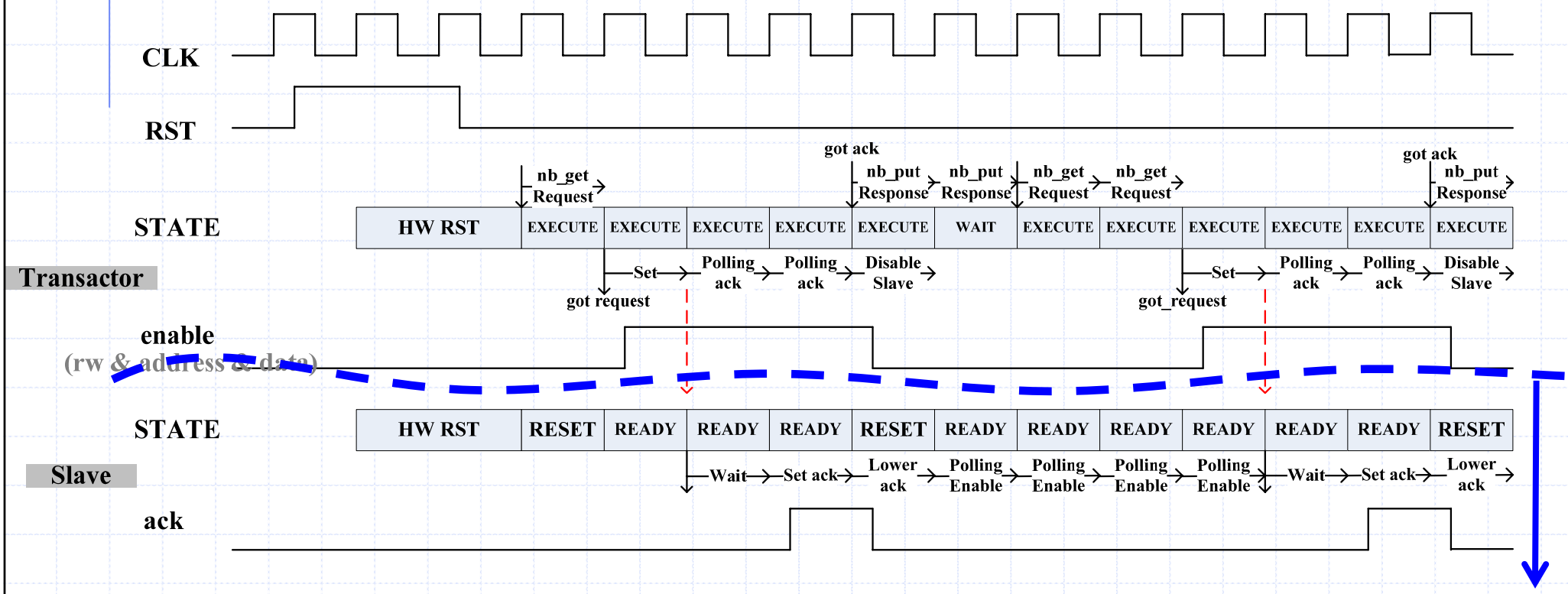




Ex:Timing Diagram

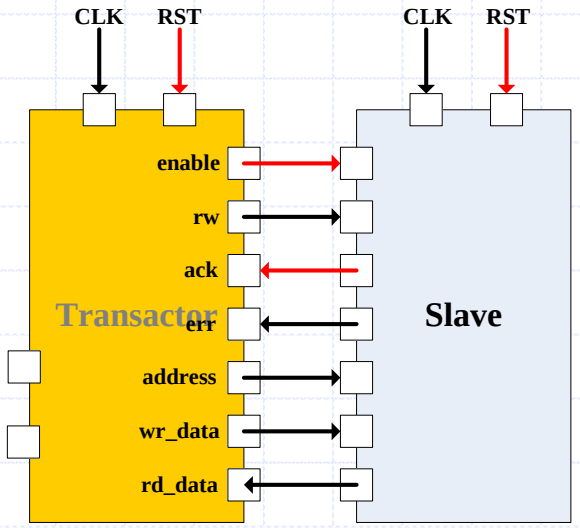
- Timing diagram of slave





Transactor

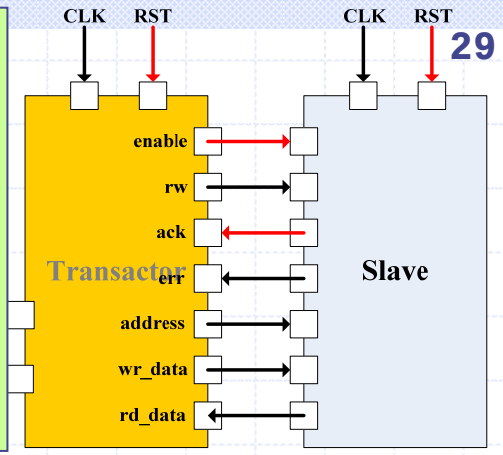
```
class transactor :  public sc_module
{
public:
transactor( sc_module_name module_name );
sc_port< tlm_nonblocking_get_if< REQUEST_TYPE > > request_port;
sc_port< tlm_nonblocking_put_if< RESPONSE_TYPE > > response_port;
SC_HAS_PROCESS( transactor );
sc_in<bool> clk;
sc_in<bool> rst;
sc_out<bool>  enable;
sc_out< bool >  rw;
sc_in<bool>   ack;
sc_in<bool>   err;
sc_out< sc_uint< ADDRESS_WIDTH> > address;
sc_out< sc_uint< DATA_WIDTH > >   wr_data;
sc_in< sc_uint< DATA_WIDTH > >   rd_data;
private:
enum state {EXECUTE , WAIT };
void run();
REQUEST_TYPE req;
RESPONSE_TYPE rsp;
bool got_request;
bool put_status;
sc_signal< state > m_state;
};
```




```
slave::slave( sc_module_name module_name ) :
sc_module( module_name ) ,
clk("clk") ,
rst("rst") ,
enable("en") ,
address("address") ,
rw("rw") ,
wr_data("wr_data" ) ,
rd_data("rd_data" ) ,
ack("ack") ,
err("err" )
{
 SC_METHOD( run );
 sensitive << clk.pos();
 dont_initialize();
}
```

Transactor (c. 1)

```
void run(){
    if(rst )
    {
      m_state = EXECUTE;
      got_request = false;
      put_status = true;
      enable = false;
    }else{
        Finite State Machine
    }
}
```




```
switch( m_state ) {
case EXECUTE :
        if( ack == true ) {
            m_state = WAIT;
            enable = false;
            got_request = false;
            if( err ) rsp.status = ERROR;
            else      rsp.status = SUCCESS;
            if( rw ) {
                rsp.type = READ;
                rsp.d = rd_data.read();
            }else
                rsp.type = WRITE;
            put_status = response_port->nb_put( rsp );
        }else{
            if( !got_request )
                got_request = request_port->nb_get( req );
            if( got_request ) {
                enable = true;
                address = req.a;
                if( req.type == WRITE ) {
                    rw = false;
                    wr_data = req.d;
                } else
                    rw = true;
            }
            break;
.....................
```

Phase 2

Phase 1

Get response from slave

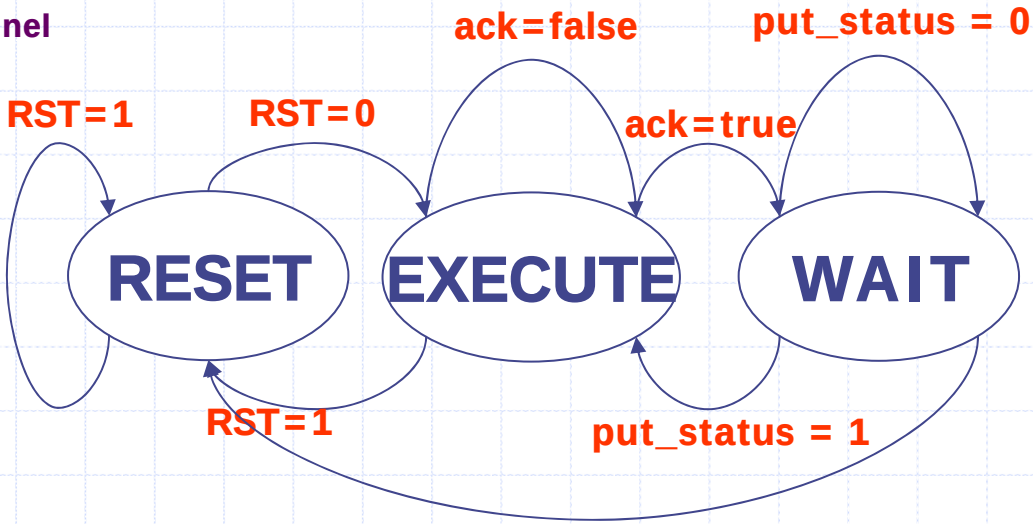
write response to transport_channel

Get request from tranport_channel

Execute next request

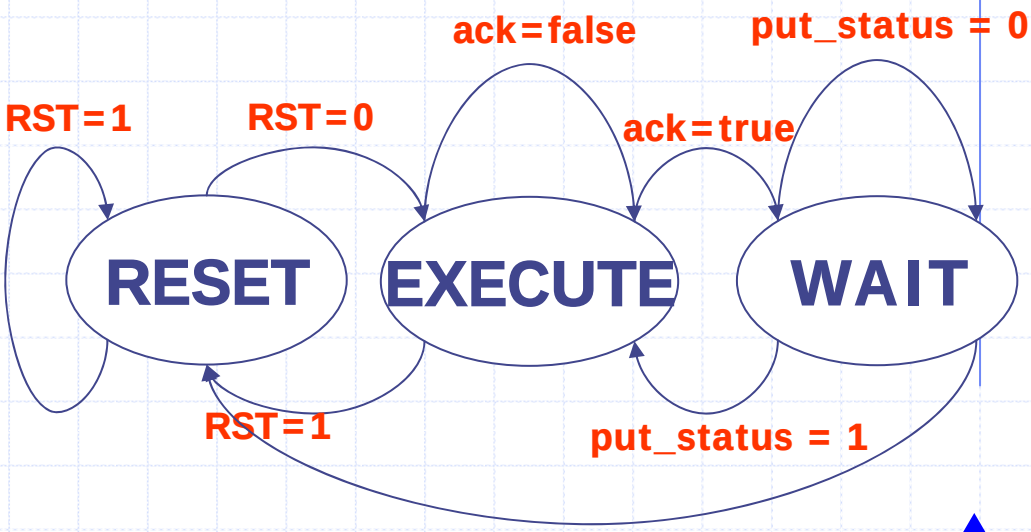
```
case WAIT :
      enable = false;
      if( !put_status ) put_status = response_port->nb_put( rsp );
      if( put_status ) m_state = EXECUTE;
      break;
default:
.....................
```

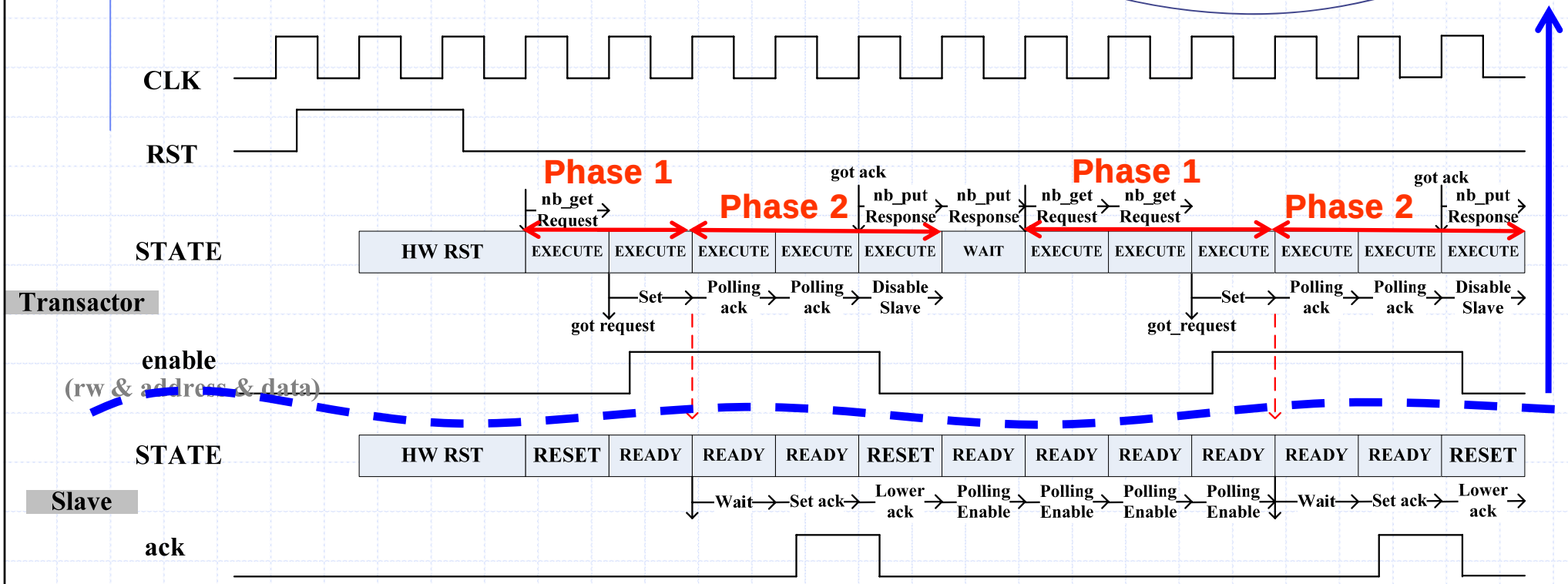
Keep trying to write response to transport_channel



Ex: Timing Diagram

- Timing diagram of transactor





RTL Master + RTL Slave

Exercises (Due 5/31)

Answer the following questions by looking into the tlm_white_paper

- Trace the code of example 3.5
- Depict the state diagram of the master part in example 3.5
- Construct the timing diagram of example 3.5
- Construct the timing diagram of example 3.4
- Compare the two timing diagrams

Typical Modeling Patterns with TLM API

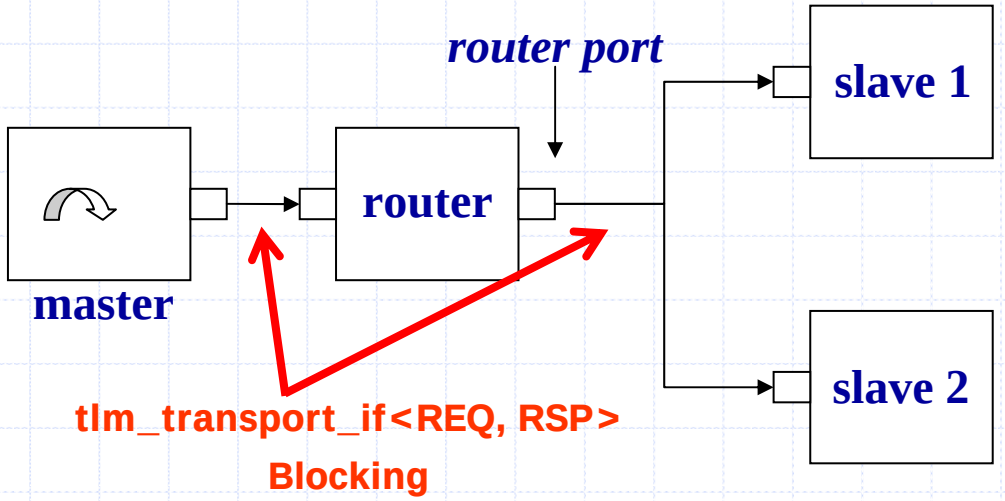
Modeling Techniques and Guidelines

- Leverage tlm_fifo to connect incompatible interfaces
  - tlm_fifo provides interfaces in blocking and non-blocking forms
  - Can do blocking puts into the fifo, and non-blocking gets out of it
- Leverage exiting TLM components as much as possible
  - tlm_fifo<T>, tlm_req_rsp_channel<REQ, RSP>, tlm_transport_channel<REQ, RSP>,
- Create generic TLM modules, channels, adapters, etc
  - Generic crossbar, pipeline, cache, parallel-to-serial adaptor
- Use deterministic modeling
  - Use two-phase TLM channels
  - Use two-phase primitive channels in SystemC
  - Employ explicit two-phase synchronization scheme

Routers

- Route the traffic generated by one master to one of the slaves
  - Receive a request from the master
  - Attempt to find an address range which contains this address
  - Return a suitable protocol error if it is not successful
  - Subtract the base address of the slave from the request
  - Forward the adjusted request to the slave
  - Send the response back to the master

```
int sc_main( int argc , char **argv ){
master m("master");
router< ADDRESS_TYPE , REQ, RSP >
                        router("router" , "master.iport.map");
mem_slave s1("slave_1");
mem_slave s2("slave_2");
m.initiator_port( router.target_port );
router.r_port( s1.target_port );
router.r_port( s2.target_port );
sc_start( -1 );
return 0;}
```



Routers (c. 1)

- RSP constructor must initialize the response to an error state
- REQ must support get_address()

```
template< typename ADDRESS , typename REQ , typename RSP>
class router :
public virtual tlm_transport_if< REQ , RSP > ,
public sc_module
{
public:
  typedef  tlm_transport_if< REQ ,RSP > if_type;
  router_port< if_type > r_port;
  sc_export<if_type> target_port;
   ..................................
  RSP transport( const REQ &req ) {
    REQ new_req = req;
    int port_index;
    if( !amap.decode( new_req.get_address() ,
                              new_req.get_address() , port_index ) )
                return RSP();

          return r_port[port_index]->transport(new_req);
  }
  ..........................
private:
  address_map<ADDRESS> amap;
  string map_file_name;
};
```

Routers (c. 2)

```
template< class ADDRESS >
class address_map
{
public:
  bool decode( const ADDRESS &before , ADDRESS &after , int &p ) const;
  void add_to_map( const ADDRESS &from , const ADDRESS &to , int p ) {
     address_range_map[p] = address_range<ADDRESS>( from , to ) ;
  }
private:        <Key, Data>
  map< int , address_range<ADDRESS> > address_range_map;
};
```

```
template<typename ADDRESS>
bool address_map<ADDRESS>::
decode( const ADDRESS &before , ADDRESS &after , int &p ) const{
  typename map< int , address_range<ADDRESS> >::const_iterator i;
  for( i = address_range_map.begin();
       i != address_range_map.end(); ++i ) {
    if( (*i).second.decode( before , after ) ) {
      p = (*i).first;
      return true;
    }
  }
  return false;
}
```

```
template< class ADDRESS >
class address_range
{
public:
  ADDRESS from , to;
  address_range( const ADDRESS &l , const ADDRESS &h ) : from( l ) , to( h ) {}
  .....
  bool decode( const ADDRESS &before , ADDRESS &after ) const {
  if( from <= before && before < to ) {
     after = before - from;
     return true;
  }
    return false;
  }
};
```

Subtract the base address

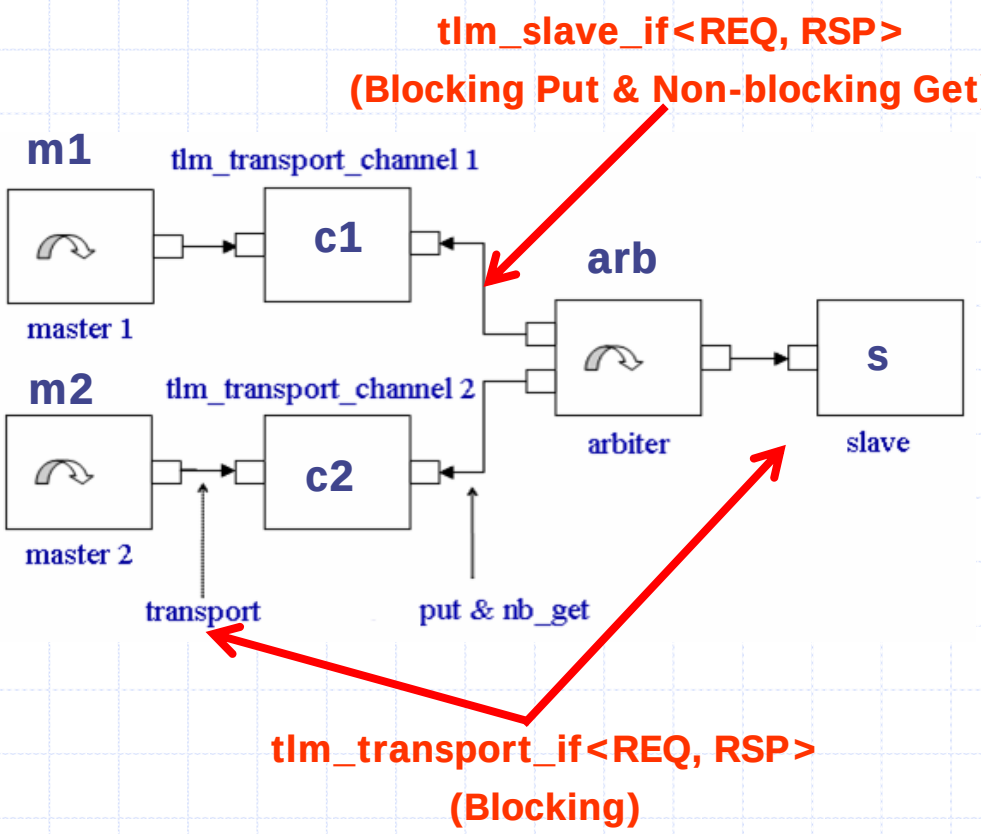
address_range_map	
address_range_map.first =Key Object (int)	address_range_map.second =Data Object (address_range)
0	from 0 to 16 (slave 1)
1	from 16 to 32 (slave 2)
2	……

- map is a sorted associative array mapping Key objects to Data objects
  - http://www.cppreference.com/cppmap/index.html; http://en.wikipedia.org/wiki/Map_(C++_container)

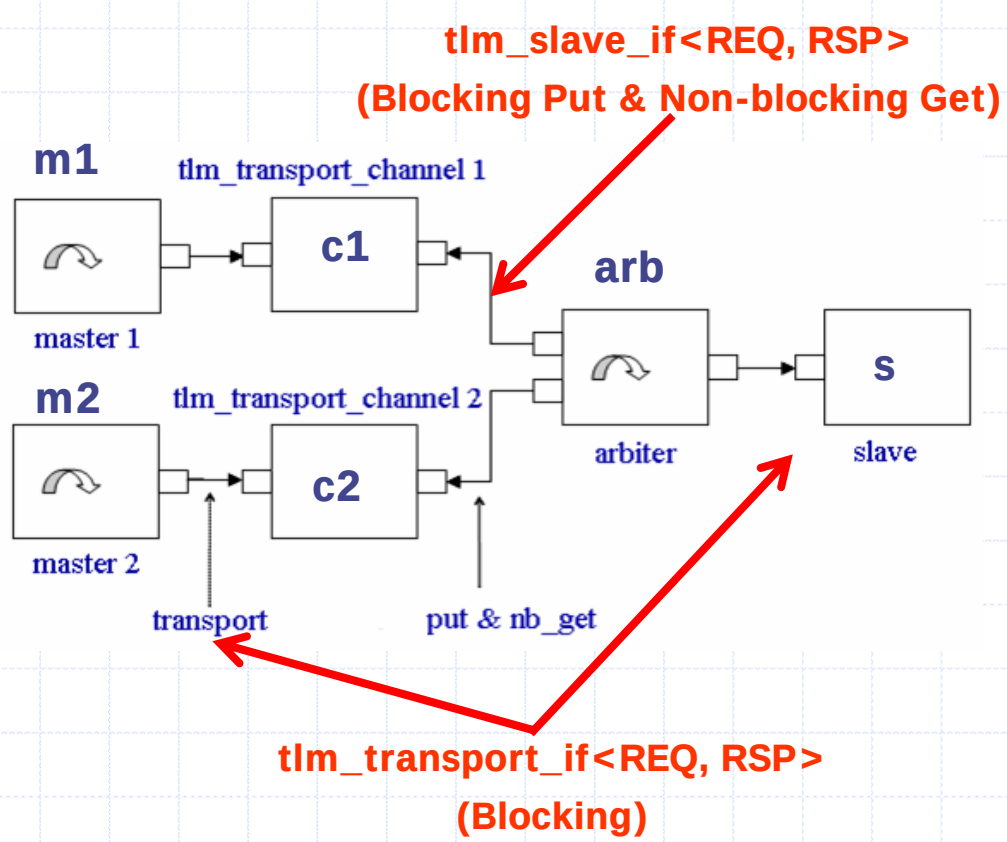
Arbiter

- Arbitrate between two simultaneous requests with timed models
- In a pure PV model, arbitration is a meaningless concept

```
class toplevel : public sc_module
{
public:
        typedef tlm_transport_channel<REQ, RSP >
                                        arb_channel_type;
        toplevel( sc_module_name module_name ) :
        sc_module( module_name ) ,
        m1("master1") ,
        m2("master2") ,
        arb("arb") ,
        s("slave") {
        m1.initiator_port( c1.target_export );
        m2.initiator_port( c2.target_export );
        arb.master_port[0]( c1.slave_export );
        arb.master_port[1]( c2.slave_export );
        arb.slave_port( s.target_port );
        arb.add_interface( &arb.master_port[0] , 3 );
        arb.add_interface( &arb.master_port[1] , 2 );  }
private:
        master m1 , m2;
        simple_arb< REQ, RSP>  arb;
        mem_slave s;
        arb_channel_type c1 , c2;
};
```



Arbiter (c. 1)



- Masters
  - Put a request into their transport channel
  - Wait for a corresponding response
- Arbiter
  - Poll all the request fifos using nb_get
  - Forward the request to the slave
  - Put the response into the response fifo in the relevant channel

Arbiter (c. 2)

```
template < typename REQ , typename RSP , int N = 2>
class simple_arb : public sc_module
{
public:
  typedef sc_port< tlm_slave_if< REQ , RSP > , 1 > port_type;

  port_type master_port[N];
  sc_port< tlm_transport_if< REQ , RSP > , 1 > slave_port;

  SC_HAS_PROCESS( simple_arb );

  simple_arb( sc_module_name module_name ,
  const sc_time &t = sc_time( 1 , SC_NS ) ) :
  sc_module( module_name ) , slave_port("slave_port") ,
  arb_t( t ) {   SC_THREAD( run );  }

  ..................................
  void add_interface( port_type *port , const int priority ) {
     if_map.insert(multimap_type::value_type(priority, port));
  }
   ..................................
}
```

```
typedef multimap< int , port_type * > multimap_type;
multimap_type if_map;
.....................

virtual void run() {
    port_type *port_ptr;
    typename multimap_type::iterator i;
    REQ req;
    RSP rsp;
    for( ;; ) {
      if( port_ptr = get_next_request( i , req ) ) {
                rsp = slave_port->transport( req );
                (*port_ptr)->put( rsp );
      }
      wait( arb_t );
    }
  }
```

```
virtual port_type *get_next_request( typename
multimap_type::iterator &i ,  REQ &req )
{
    port_type *p;
    for( i = if_map.begin(); i != if_map.end(); ++i ) {
      p = (*i).second;
      if( (*p)->nb_get( req )  )
          return p;
     }
    return 0;
}
```

Always start from the highest priority

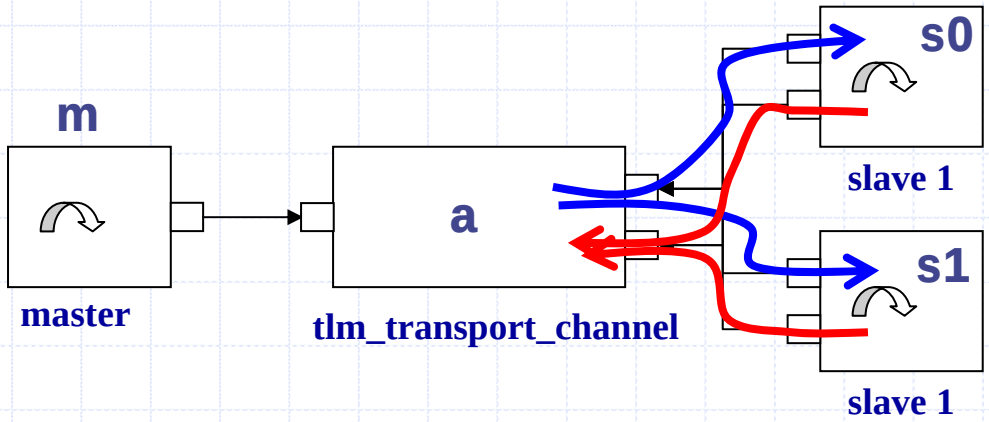
Arbiter (c. 3)

- Multimap (in stl library)
  - A one to many mapping from priority level to port
  - Can be configured and reconfigured at any time
  - Come with many access functions useful for arbitration
    - Easy to find all ports of the same priority level for use in a prioritized round robin arbitration scheme

Decentralized Routing

- Centralized routing may diverge too far from the implementation
- Decentralized routing
  - All of the slaves monitor all of the requests coming in using peek
  - Pop the request out of the request fifo using get
  - Processes the request and sends a response to the response fifo

```
typedef tlm_transport_channel< REQ , RSP> channel_type;
int sc_main( int argc , char **argv ){
master m("master");
mem_slave s0("even_slave" , 0 );
mem_slave s1("odd_slave" , 1 );
channel_type a;  m.initiator_port( a.target_export );
s0.request_port( a.get_request_export );
s0.response_port( a.put_response_export );
s1.request_port( a.get_request_export );
s1.response_port( a.put_response_export );
sc_start( -1 );
return 0;}
```



Decentralized Routing (c. 1)

```
class mem_slave :  public sc_module
{
public:
  mem_slave( sc_module_name module_name , int r );
  SC_HAS_PROCESS( mem_slave );
   sc_port<tlm_get_peek_if <REQ, RSP> request_port;
   sc_port<tlm_blocking_put_if <REQ, RSP> response_port;
   ~mem_slave();private:
    void run();
    bool decode( const ADDRESS_TYPE &a );
    RSP process_request( const REQ &req );
    ...................
};
```

```
void mem_slave::run()
{
basic_request<ADDRESS_TYPE,DATA_TYPE> req;
while( true ) {
      request_port->peek( req );
     if( decode( req.a ) ){
          basic_response<DATA_TYPE> rsp;
          request_port->get();
          rsp = process_request( req );
          response_port->put( rsp );
     }
    wait( request_port->ok_to_get() );
}
}
```

Pipeline

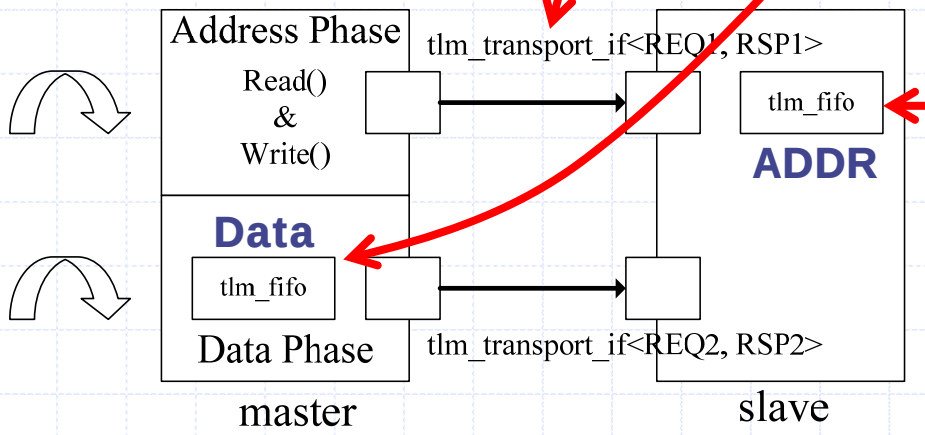
- Separate the data transfer into address phase and data phase

```
int sc_main( int argc , char **argv ) {
  pipelined_slave s("slave" , 10 , 2 );
  pipelined_master m("master");
  m.address_port( s.address_export );
  m.data_port( s.data_export );
```

```
void pipelined_master::address_phase()
{
   for( int i = 0; i < 20; i++ ) {
      initiate_write( i , i + 50 );
      initiate_read( i );
   }
}
```

```
bool pipelined_master::initiate_write(const ADDRESS_TYPE &a,
                                      const DATA_TYPE &d)
{
address_phase_request<ADDRESS_TYPE> address_req;
data_phase_request<DATA_TYPE> data_req;
address_req.type = data_req.type = WRITE;
address_req.a = a;
if( address_port->transport( address_req ) != ADDRESS_OK )
return false;
outstanding.put( data_req );
return true;
}
```

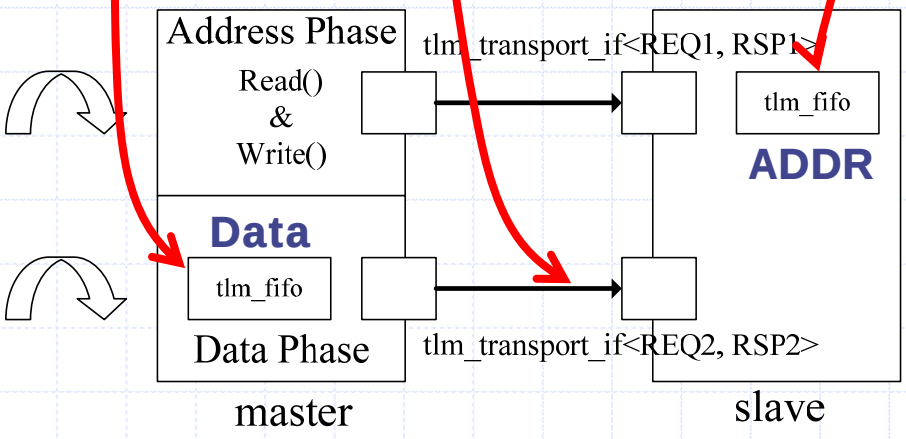
```
RSP1
transport(REQ1) {
pipeline->put( req );
return address_phase_status( ADDRESS_OK );
}
```



Pipeline (c. 1)

- Data phase

```
void pipelined_master::data_phase() {
data_phase_request<ADDRESS_TYPE> data_req;
data_phase_response<DATA_TYPE> data_rsp;
while( true ) {
data_req = outstanding.get();
data_rsp = data_port->transport( data_req );
}
}
```

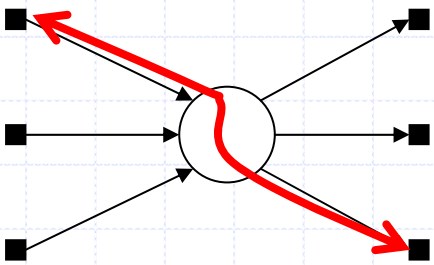


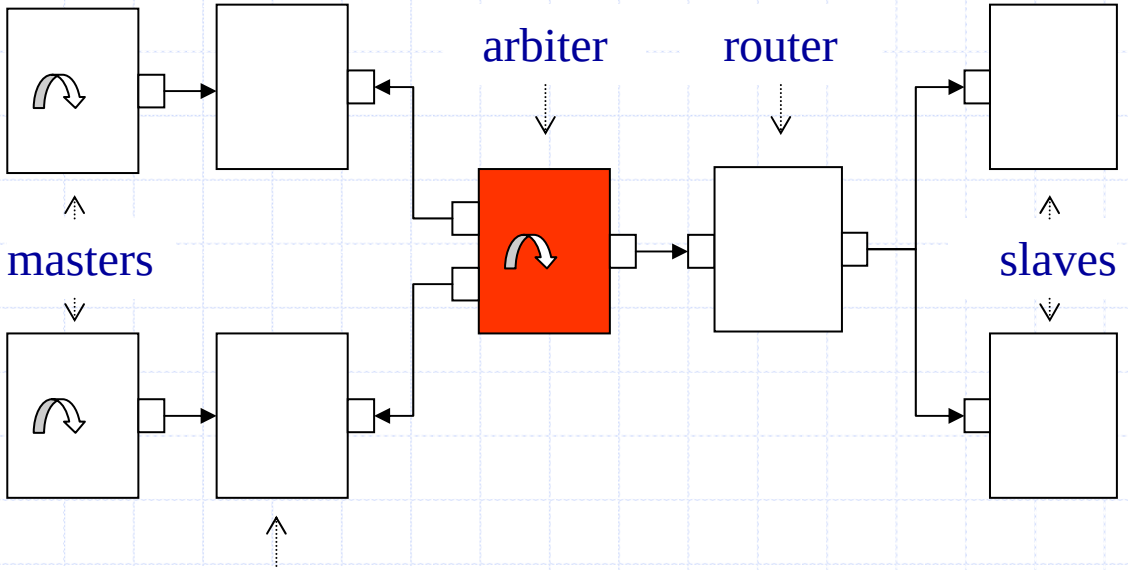

```
RSP transport(REQ)
data_phase_response<DATA_TYPE > rsp;
address_phase_request< ADDRESS_TYPE > pending;
  while( pipeline->nb_can_put() )
       wait( pipeline->ok_to_get() );

 pipeline->nb_peek( pending );
  if( pending.type != req.type ) {
     rsp.status = DATA_ERROR;
     return rsp;
   }
   pipeline->get();
   rsp.status = DATA_OK;
   rsp.type = req.type;
   switch( req.type ) {
       case READ :
            rsp.rd_data = memory[pending.a];
            break;
       case WRITE :
            memory[pending.a] = req.wr_data;
            break;
        default :    assert( 0 );
     }
    return rsp;
}
```

Architectural Exploration Using TLM API

Hub and Spoke

- All transactions pass through a central hub
- The hub arbitrates between the various requests
- Not efficient in terms of throughput
- Efficient in terms of area and cost
- Only 1 arbiter is needed





Cross Bar Switch

- Being able to make more than one connections at the same time
- No central arbitration
- Every slave has to arbitrate between all the masters
- More expensive and less predictable
- Better throughput