System Level Design (and Modelling for Embedded Systems)

12 – Transaction Level Modelling (Part 1)

Kim Grüttner <kim.gruettner@dlr.de>
Jörg Walter <joerg.walter@offis.de>
Henning Schlender <henning.schlender@dlr.de>
Sven Mehlhop <sven.mehlhop@offis.de>

Distributed Computation and Communication, R&D Division Manufacturing
OFFIS – Institute for Information Technology

Institute of Systems Engineering for Future Mobility
German Aerospace Center (DLR-SE)

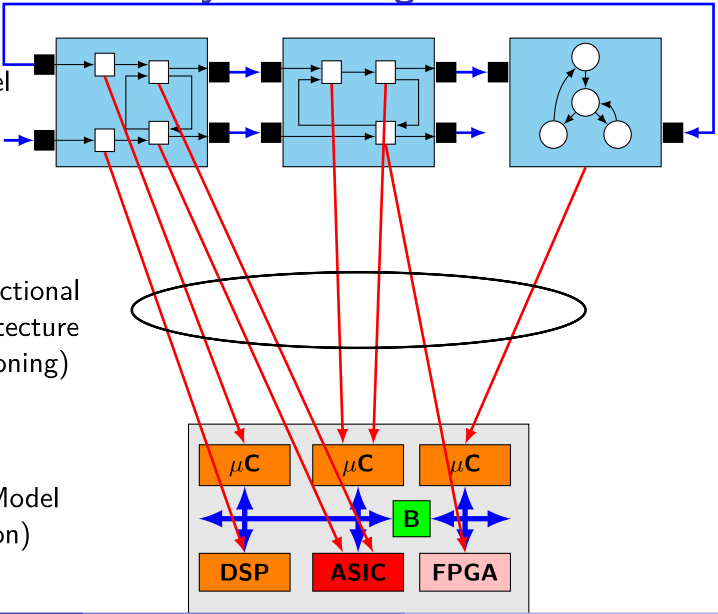
Based on the slides of M. Radetzki 2005–2008
University of Stuttgart

Models for embedded System Design

Functional Model
(Specification)

Mapping of Functional
Blocks to Architecture
Blocks (Partitioning)

Architecture Model
(Specification)



SystemC Modelling Levels

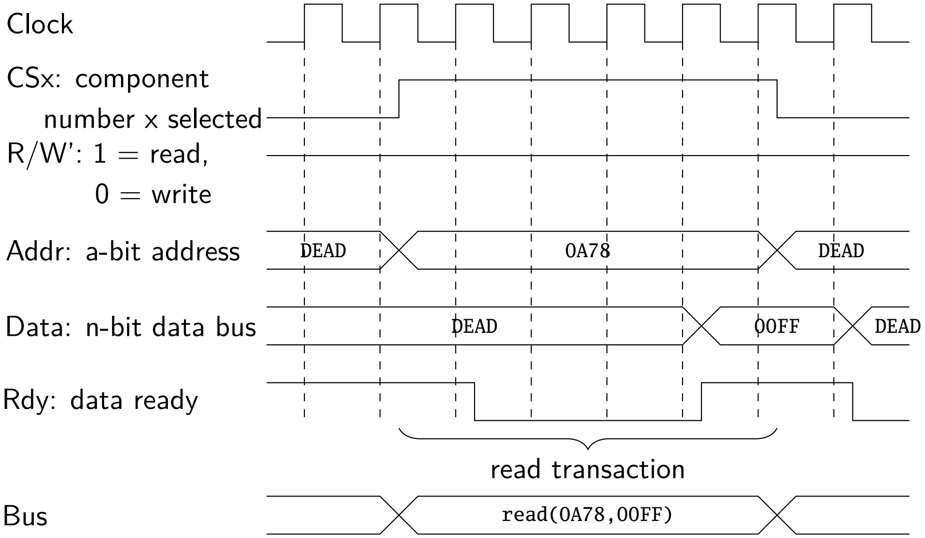
- **Functional View (FV)**
  - Concurrent processes
  - Communication mechanisms, e.g. FIFO channels

Functional models

Architecture models

- **Programmers View (PV)**
  - Modelling of the (bus-based architecture with no or little (PV+T) consideration of timing aspects
- **Architecture View**
  - Cycle approximate (CX) modelling of bus transactions
- **Verification View**
  - Cycle accurate (CA) modelling of bus transactions

Bus Protocol at RTL (Implement.)



Transaction Level Modelling

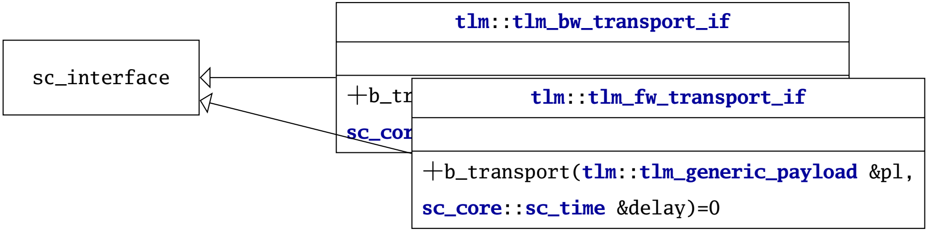
- **Modelling bus transactions as function calls instead of signal protocols**
  - `payload pl; sc_time delay;`
  - `pl.set_read()` : read transaction
  - `b_transport(pl,delay)` : perform transaction
- **Higher simulation speed compared to RTL**
  - no signals, no (or fewer) events
- **Less modelling effort compared to RTL**
  - shorter source code, less modelling detail
- **Addresses the three architecture modelling views:**
  - Programmers View (PV)
  - Architecture View (CX)
  - Verification View (CA)

Aspects to be Modelled

- **Master**
  - can take ownership of the bus
  - can initiate transactions
  - e.g.: CPU is a bus master
- **Arbitration**
  - in case of multiple masters
  - handles parallel transactions
  - decides which master gets the bus
- **Slave**
  - is a target of transactions
  - responds to transactions
  - e.g. RAM is a bus slave
- **Address decoding/mapping**
  - in case of multiple slaves
  - decodes the bus address
  - decides which slave belongs to the address
- **Transactions**
  - define the data transfer operations supported by the bus
    - e.g. `b_transport()` performing read/write depending on payload attributes
  - used (required) by masters, provided by slaves

Transactions

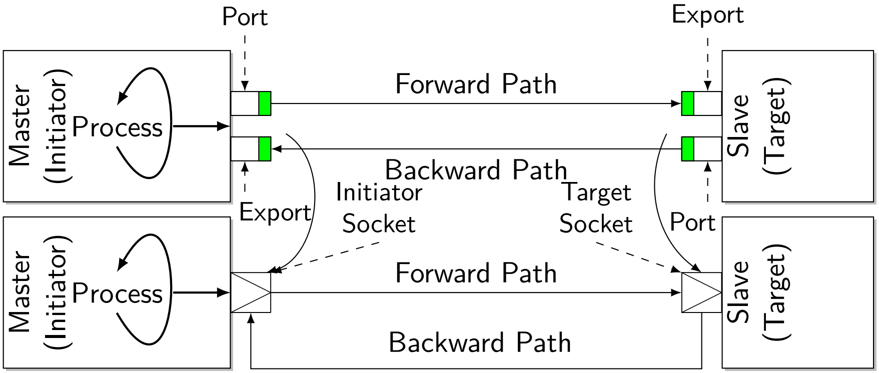
Possible transactions are defined by the payload attributes.



```
class tlm_generic_payload
{
public:
  void set_read();
  void set_write();
  void set_data_ptr(unsigned char *ptr);
  void set_address(const sc_dt::uint64 addr);
  ...
};
```

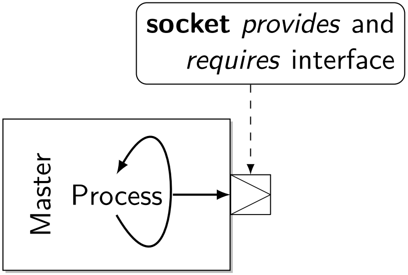
Socket

- Provides a **port** and an **export**
- Separation in **initiator** and **target**
- Initiator initiates transactions via **initiator socket**
- Target recieves transaction via **target socket**
- **Forward** and **backward path** for non-blocking transactions



Master

- Masters use (**require**) the forward transport interface and **provide** (implement) backward transport interface
- Masters have a **socket** via which they initiate transactions



```
SC_MODULE(master)
: private tlm::tlm_bw_transport_if<>
{
  tlm::tlm_initiator_socket<>
      initiator_socket;
  void action();
  unsigned start, end;
  SC_HAS_PROCESS(master);
  master( sc_module_name
      , unsigned start_addr
      , unsigned end_addr );
  ...
};
```

Master Implementation

- **Example:**
  - Master writes random data to a range of addresses
  - Master reads data from the same range of addresses
  - Address range is defined by constructor parameters

```
// constructor
master::master( sc_core::sc_module_name
              , unsigned start_addr
              , unsigned end_addr )
  : initiator_socket("socket"),
    start(start_addr), end(end_addr)
{
  SC_THREAD(action);
}
```

Master Process: action()

```
tlm::tlm_generic_payload pl; unsigned data;
sc_core::sc_time delay = SC_ZERO_TIME;
wait(10, SC_NS);
for (unsigned toggle = 0; toggle < 2; toggle++) {
  if (toggle == 0) pl.set_write();
  else pl.set_read();
  for (unsigned addr = start; addr <= end; ++addr) {
    data = rand();
    pl.set_data_ptr(reinterpret_cast<unsigned char*>(&data));
    pl.set_address(addr);
    initiator_socket->b_transport(pl, delay);
    cout << name() << " transport: addr=" << addr << ", "
      << "data=" << data << ", read=" << pl.get_read()
      << " at " << sc_time_stamp() << endl;
    wait(1, SC_NS);
  }
}
```

transaction via socket

Slave (e.g.: a RAM)

Slaves use (**require**) the backward transport interface and **provide** (implement) the forward transport interface.
Slaves have a **target socket** via which they recieve transactions.

```
struct ram : public sc_module
           , public tlm::tlm_fw_transport_if<>
{
  tlm::tlm_target_socket<> target_socket;
  ram( sc_module_name name, unsigned size );

  virtual void b_transport( tlm::tlm_generic_payload& trans
                          , sc_core::sc_time& delay );
private: // implementation details
  std::vector<unsigned> mem;
};
```

RAM Constructor

```
ram::ram( sc_module_name /* unused */
        , unsigned size )
: target_socket( "target_socket" )
, mem( size )
{
  target_socket.bind(*this);
}
```

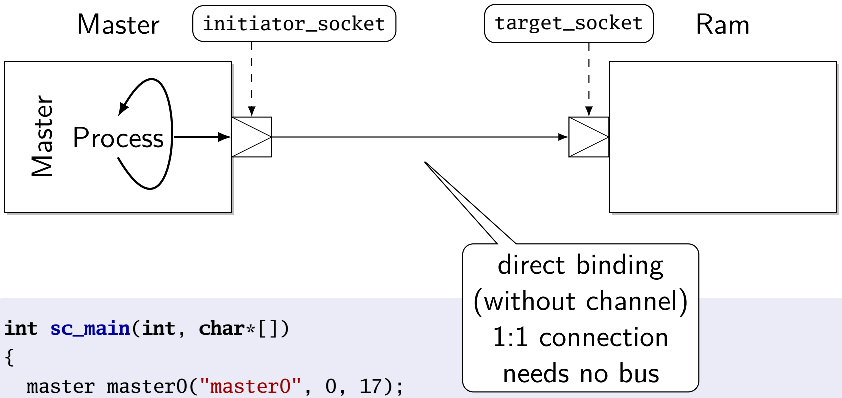
bind itself to socket

- **Explicit socket:**
  - common modelling technique
  - hides implementation detail, that module itself implements interface (could be an internal channel instead)
  - multiple interfaces can be provided at the same time

RAM Slave Transaction Implementation

```
void ram::b_transport( tlm::tlm_generic_payload& trans
                     , sc_core::sc_time& )
{
  unsigned& data = *reinterpret_cast<unsigned*>(trans.get_data_ptr());
  sc_dt::uint64 addr = trans.get_address();
  if (check_address(addr)) { // test for wrong address
    trans.set_response_status(tlm::TLM_ADDRESS_ERROR_RESPONSE);
    return; }
  // ... further checks on payload object
  switch ( trans.get_command() ){
    case tlm::TLM_WRITE_COMMAND:
      mem[addr] = data; break;
    case tlm::TLM_READ_COMMAND:
      data = mem[addr]; break; }
  trans.set_response_status(tlm::TLM_OK_RESPONSE);
};
```

The Bus System



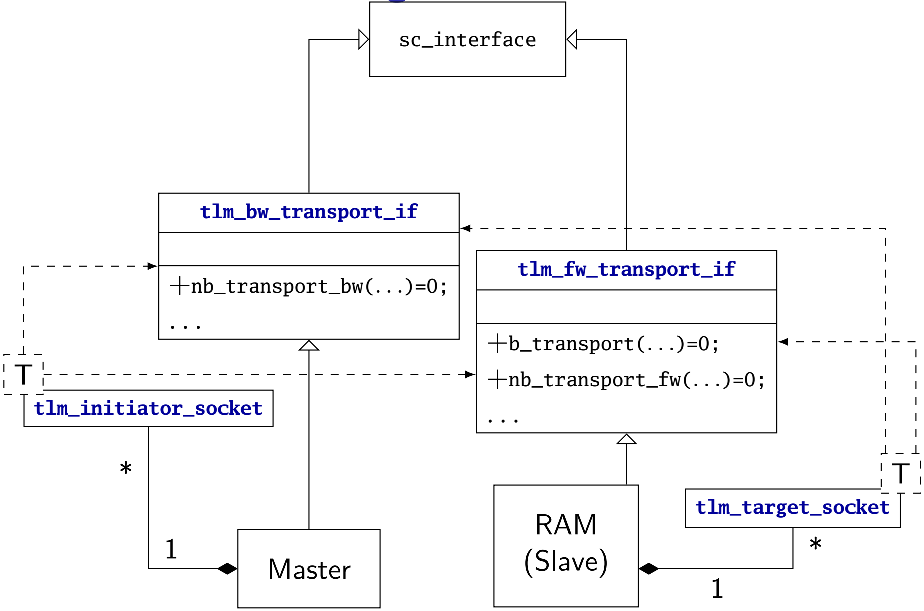
```
int sc_main(int, char*[])
{
  master master0("master0", 0, 17);
  ram    ram0   ("ram0"   , 16);
  master0.initiator_socket( ram0.target_socket );
  sc_start();
  return 0;
}
```

Simulation Output

```
master1 write(0, 0) at 10 ns
master1 write(1, 1481765933) at 11 ns
master1 write(2, 1085377743) at 12 ns
master1 write(3, 1270216262) at 13 ns
master1 write(4, 1191391529) at 14 ns
master1 write(5, 812669700) at 15 ns
master1 write(6, 553475508) at 16 ns
master1 write(7, 445349752) at 17 ns
master1 write(8, 1344887256) at 18 ns
master1 write(9, 730417256) at 19 ns
master1 write(10, 1812158119) at 20 ns
master1 write(11, 147699711) at 21 ns
master1 write(12, 880268351) at 22 ns
master1 write(13, 1889772843) at 23 ns
master1 write(14, 686078705) at 24 ns
master1 write(15, 2105754108) at 25 ns
ram1 write(16, __) : ERROR address out of
range
master1 write(16, 182546393) at 26 ns
ram1 write(17, __) : ERROR address out of
range
master1 write(17, 1949118330) at 27 ns
```

```
master1 read(0, 0) at 28 ns
master1 read(1, 1481765933) at 29 ns
master1 read(2, 1085377743) at 30 ns
master1 read(3, 1270216262) at 31 ns
master1 read(4, 1191391529) at 32 ns
master1 read(5, 812669700) at 33 ns
master1 read(6, 553475508) at 34 ns
master1 read(7, 445349752) at 35 ns
master1 read(8, 1344887256) at 36 ns
master1 read(9, 730417256) at 37 ns
master1 read(10, 1812158119) at 38 ns
master1 read(11, 147699711) at 39 ns
master1 read(12, 880268351) at 40 ns
master1 read(13, 1889772843) at 41 ns
master1 read(14, 686078705) at 42 ns
master1 read(15, 2105754108) at 43 ns
ram1 read(16, __) : ERROR address out of
range
master1 read(16, 2105754108) at 44 ns
ram1 read(17, __) : ERROR address out of
range
master1 read(17, 2105754108) at 45 ns
```

TLM Model Class Diagram



Convenience Sockets

Class	Register callbacks?	Multi-ports?	b/nb conv.?	Tagged?
`tlm_initiator_socket`	no	yes	-	no
`tlm_target_socket`	no	yes	no	no
`simple_initiator_socket`	yes	no	-	no
`simple_initiator_socket_tagged`	yes	no	-	yes
`simple_target_socket`	yes	no	yes	no
`simple_target_socket_tagged`	yes	no	yes	yes
`passthrough_target_socket`	yes	no	no	no
`passthrough_target_socket_tagged`	yes	no	no	yes
`multi_passthrough_initiator_socket`	yes	yes	-	yes
`multi_passthrough_target_socket`	yes	yes	no	yes

Convenience Sockets Comp.

TLM Target Socket Usage:

```
struct Target: sc_module, tlm::tlm_fw_transport_if<> {
  tlm::tlm_target_socket<> socket;

  SC_CTOR(Target) : socket("socket") {
    socket.bind( *this );
  }
  virtual tlm::tlm_sync_enum nb_transport_fw(
      tlm::tlm_generic payload& trans
    , tlm::tlm_phase& phase
    , sc_core::sc_time& t)
  { ... }

  virtual void b_transport( tlm::tlm_generic_payload& trans,
                            sc_core::sc_time& delay )
  { ... }

  ... // All other methods of tlm::tlm_fw_transport_if<>
};
```

Convenience Sockets Comp.

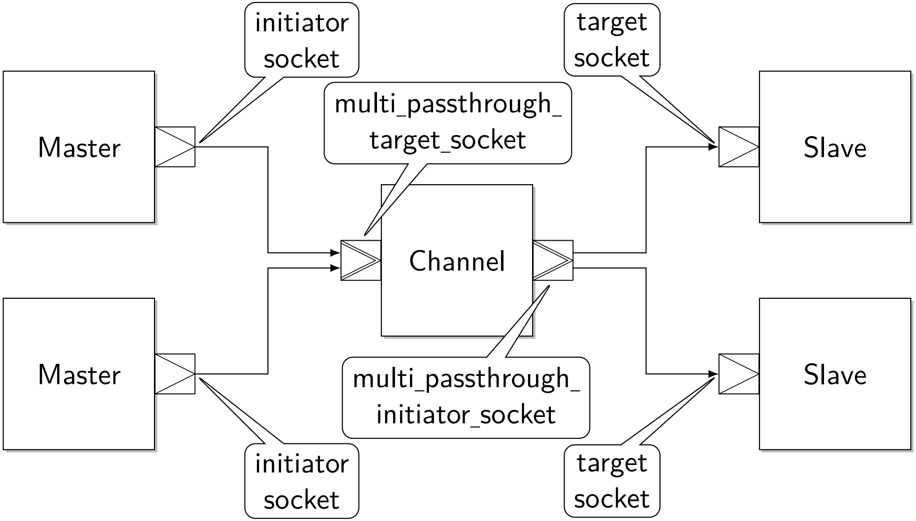
Simple Target Socket Usage:

```
struct Target: sc_module {
  tlm_utils::simple_target_socket<Target> socket;

  SC_CTOR(Target) : socket("socket") {
    socket.register_b_transport( this, &Target::b_transport );
  }

  virtual void b_transport( tlm::tlm_generic_payload& trans,
                            sc_core::sc_time& delay)
  { ... }
};
```

Multiple Masters and Slaves

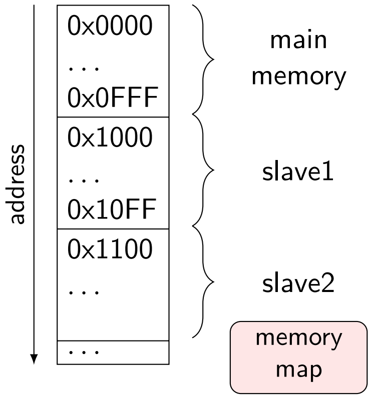


PV: arbitration not modelled; transactions can occur simultaneously

Address decoding (memory-mapped I/O) needs to be modelled

Memory-Mapped IO

- bus slaves are mapped into the bus address space
- access by transactions via the bus
- bus (address) decoder detects which slave is addressed and sends the transactions to this slave



modelling options:

- each slave knows its address range; each slave peeks addresses on the bus and responds when an address from its range occurs
- a decode (part of bus system) determines which slave is addressed and sends the transaction only to this slave

Bus Channel

```
struct bus : public sc_module
{
  tlm_utils::multi_passthrough_target_socket<bus> target_socket;
  tlm_utils::multi_passthrough_initiator_socket<bus> init_socket;

  SC_CTOR(bus): target_socket("target_bus_socket"),
                init_socket("initiator_bus_socket")
     { target_socket.register_b_transport(this, &bus::b_transport;}

private:
  // implement bus_if
  void b_transport( int id, tlm::tlm_generic_payload& trans,
                          sc_core::sc_time& delay );
  // stuff for address decoding
  void end_of_elaboration();
  unsigned decode(unsigned addr);
  std::vector<unsigned> starts, ends;
};
```

Reading Memory Map From File

```
void bus::end_of_elaboration() {
  unsigned i, slave_start, slave_end;
  ifstream mem_map("mem_map.txt");
  unsigned slaves = init_socket.size();
  starts.resize( slaves );
  ends.resize( slaves );
  while( mem_map >> dec >> i ) {
    mem_map >> hex >> slave_start >> slave_end;
    if( i >= slaves ) {
      cout << "Bus ERROR: slave number does not exist" << endl;
      sc_stop();
    } else {
      starts[i] = slave_start;
      ends[i]   = slave_end;
      cout << "Bus slave " << i << " starts "
           << slave_start  << " ends " << slave_end << endl;
    }
  }
}
```

mem_map.txt

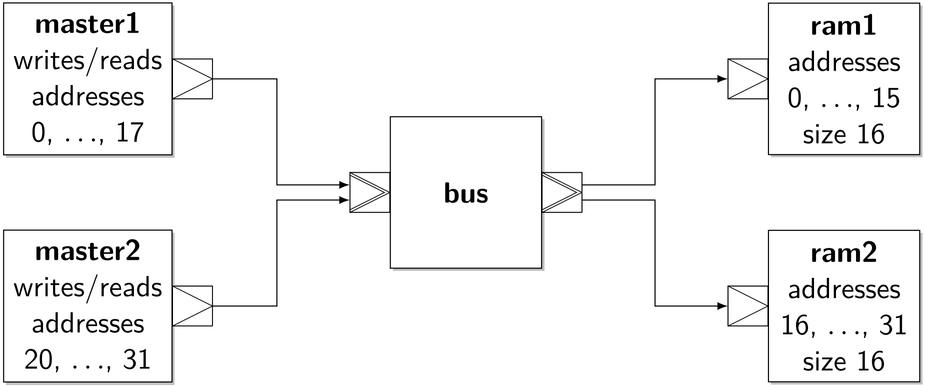
```
slave   start   end
 0      0x00    0x0F
 1      0x10    0x1F
```

Address Decoding and Transaction Forwarding

```
void bus::b_transport(int, tlm::tlm_generic_payload &trans,
  sc_core::sc_time &delay)
{
  sc_dt::uint64 global_addr = trans.get_address();
  int slave_id = decode(global_addr);
  sc_dt::uint64 slave_addr = global_addr-starts[slave_id];
  trans.set_address(slave_addr);
  init_socket[slave_id]->b_transport(trans, delay);
  trans.set_address(global_addr);
}
unsigned bus::decode(unsigned addr)
{
  for( int i=0; i < init_socket.size(); i++ )
    if( (addr >= starts[i]) && (addr <= ends[i]) )
      return i;
  cout << "Bus ERROR: address " << addr
       << " not found" << endl;:
  return 0;
}
```

	starts	ends
0	0x00	0x0F
1	0x10	0x1F

Example System



Example System

```
int sc_main( int, char*[] )
{
  // instantiate system components
  master master1("master1",  0, 17);
  master master2("master2", 20, 31);
  ram    ram1   ("ram1"   , 16);
  ram    ram2   ("ram2"   , 16);
  bus    bus1   ("bus1");

  // connect system interfaces
  master1.initiator_socket(bus1.target_socket);
  master2.initiator_socket(bus1.target_socket);
  bus1.init_socket(ram1.target_socket);
  bus1.init_socket(ram2.target_socket);

  sc_core::sc_start();
  return 0;
}
```

Simulation Output

```
Bus slave 0 starts 0 ends 15
Bus slave 1 starts 16 ends 31
master1 write(0, 0) at 10 ns
master2 write(20, 1481765933) at 10 ns
master1 write(1, 1085377743) at 11 ns
master2 write(21, 1270216262) at 11 ns
master1 write(2, 1191391529) at 12 ns
master2 write(22, 812669700) at 12 ns
master1 write(3, 553475508) at 13 ns
master2 write(23, 445349752) at 13 ns
master1 write(4, 1344887256) at 14 ns
master2 write(24, 730417256) at 14 ns
master1 write(5, 1812158119) at 15 ns
master2 write(25, 147699711) at 15 ns
master1 write(6, 880268351) at 16 ns
master2 write(26, 1889772843) at 16 ns
master1 write(7, 686078705) at 17 ns
master2 write(27, 2105754108) at 17 ns
master1 write(8, 182546393) at 18 ns
master2 write(28, 1949118330) at 18 ns
```

```
master2 read(27, 2105754108) at 29 ns
master1 read(2, 1191391529) at 30 ns
master2 read(28, 1949118330) at 30 ns
master1 read(3, 553475508) at 31 ns
master2 read(29, 1979932169) at 31 ns
master1 read(4, 1344887256) at 32 ns
master2 read(30, 1873226917) at 32 ns
master1 read(5, 1812158119) at 33 ns
master2 read(31, 1486937972) at 33 ns
master1 read(6, 880268351) at 34 ns
master1 read(7, 686078705) at 35 ns
master1 read(8, 182546393) at 36 ns
master1 read(9, 220137366) at 37 ns
master1 read(10, 1089957932) at 38 ns
master1 read(11, 715669847) at 39 ns
master1 read(12, 1196032868) at 40 ns
master1 read(13, 777206980) at 41 ns
master1 read(14, 68706223) at 42 ns
master1 read(15, 1843638549) at 43 ns
master1 read(16, 212567592) at 44 ns
master1 read(17, 1883488164) at 45 ns
```

Timing Considerations

- **Programmers View**
  - Simultaneous transactions from multiple masters
  - No arbitration modelled
  - Transactions take no time; no timing in bus
  - System timing only if modelled in the masters/slaves

- **Architecture View**
  - Only one transaction at a time
  - Arbitration modelled
  - Transactions time is estimated; modelled in bus
  - Influence of bus conflicts on system timing modelled; additional timing aspects in the masters/slaves

add arbitration and bus timing
better accurancy of the model
lower simulation performance