Chapter 7

Transaction Level Modeling (TLM)

Zainalabedin Navabi

Transaction Level Modeling

- Introduction
- + Processing Elements
- + Data
- + Abstract Communications
- + Complete System

Transaction Level Modeling

- Introduction
- Processing Elements
  - Initiators
  - Targets
  - Interconnects
- Data
  - Generic Payload
- Abstract Communications
  - + Coding Styles
  - + TLM-2.0 Transport Interfaces
- Complete System
  - Blocking Example
  - Non-blocking Example

Transaction Level Modeling

- Introduction
- + Processing Elements
- + Data
- + Abstract Communications
- + Complete System

Electronic System Level (ESL)

- An abstraction level higher than RTL
  - Little implementation details
  - Abstract timing
- Appropriate for today's complex systems
- An entry level for simulation, synthesis, and test

Transaction Level Modeling (TLM)

- Means of design at ESL
- Separates communication from computation within a system
- Communications are performed through function calls
- A transaction is the data transfer between two modules

Why TLM?

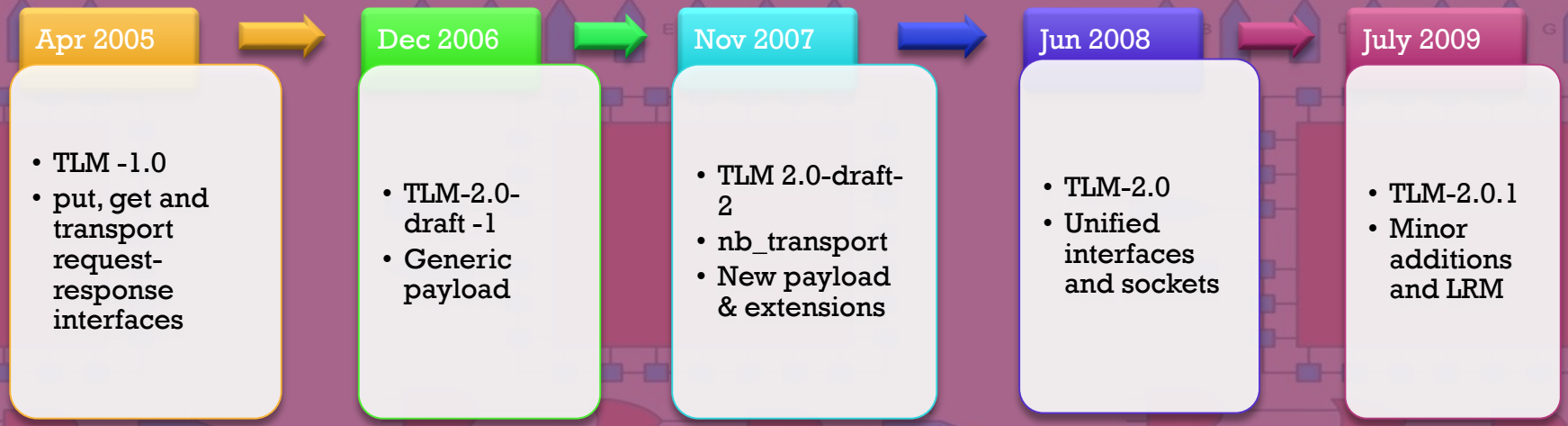
- Higher simulation speed than RTL
- Design space exploration
- Early verification
- HW/SW communication

OSCI TLM-2.0 Standard

- The Generic Payload
- Sockets
  - Initiator and Target Sockets
    - Simple Sockets
    - Tagged Sockets
    - Multi Sockets
- TLM-2.0 Core Interfaces
  - Transport interfaces
    - Blocking transport interface
    - Non-blocking transport interface
  - Direct memory interface
  - Debug transport interface
- Phases and Base Protocol

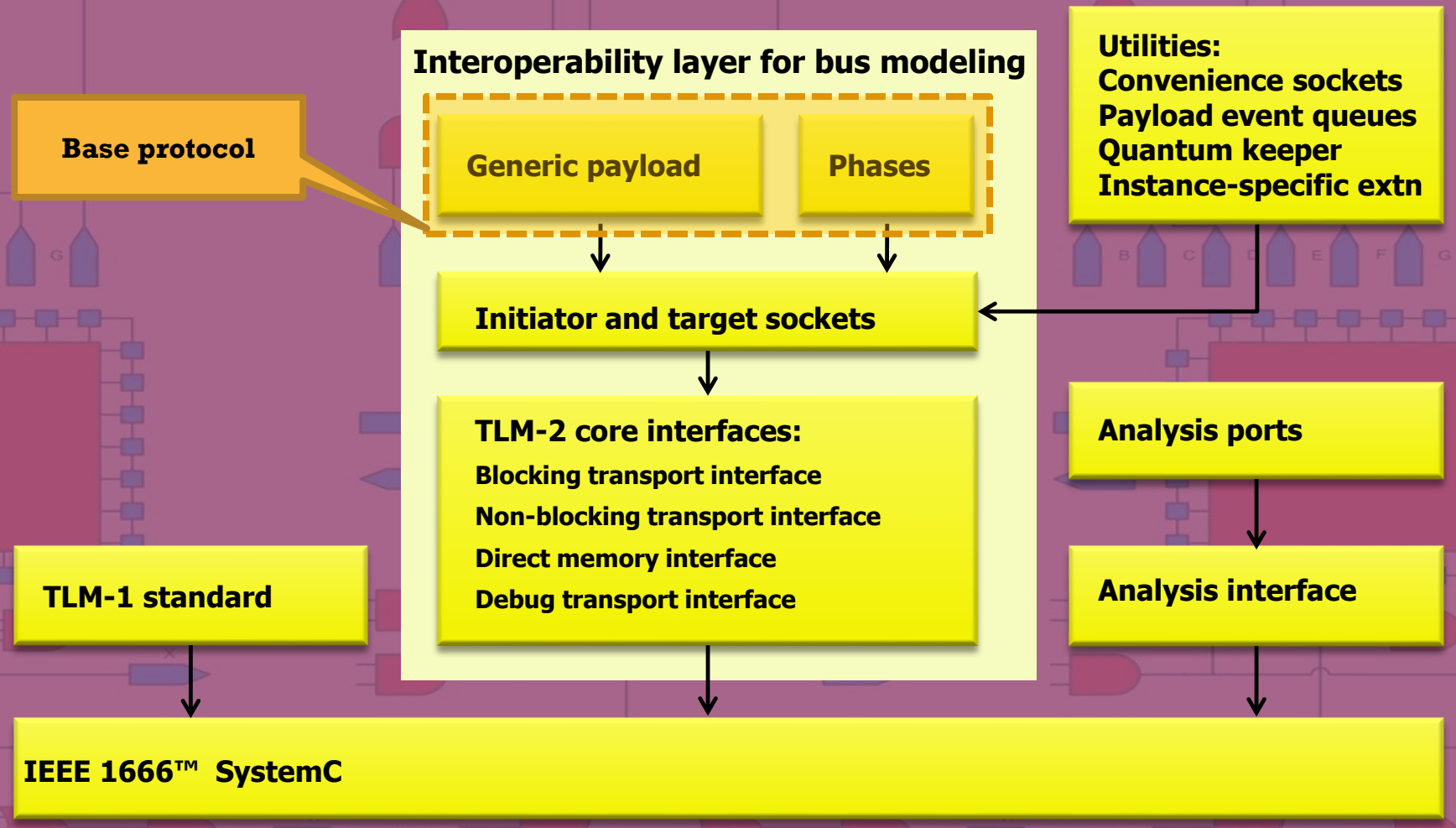
Such as RTL components: Bus, Register, …

OSCI TLM Development



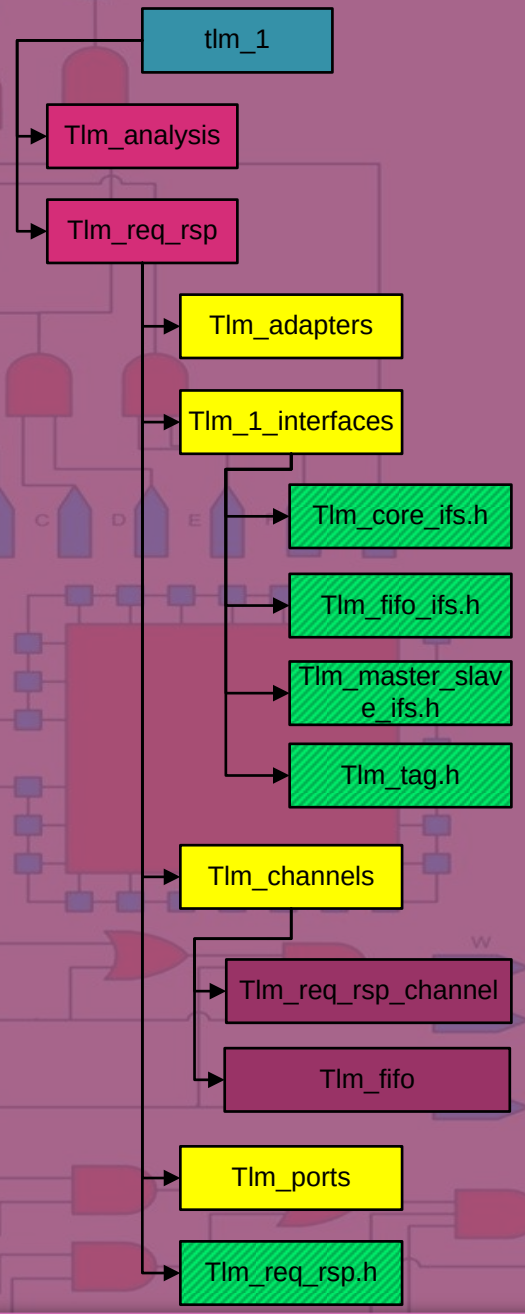
* Open SystemC Initiative, 31 May 2009

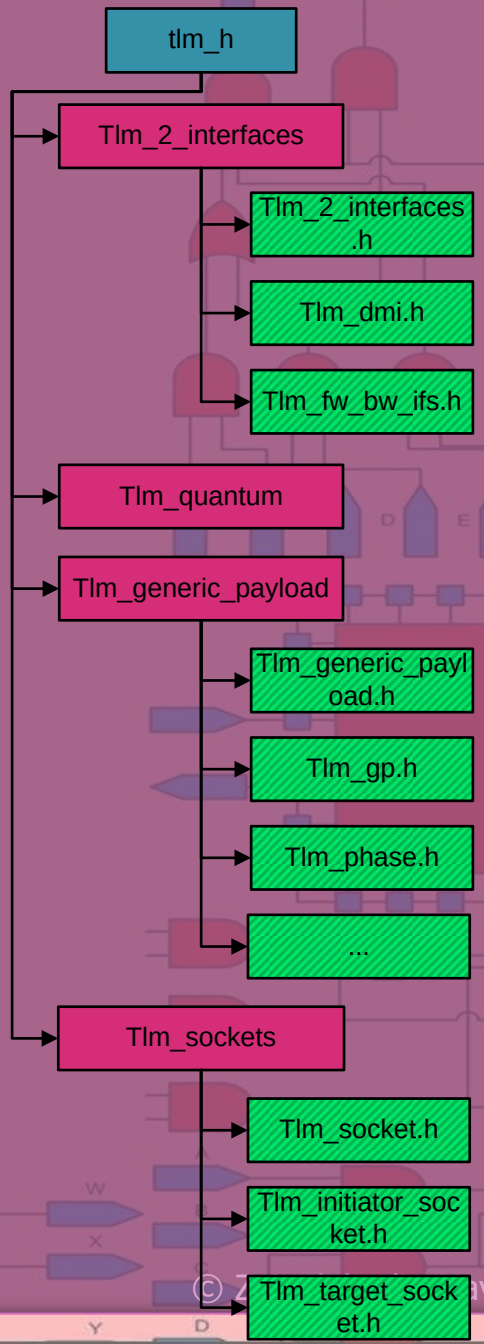
TLM-2.0 library Structure

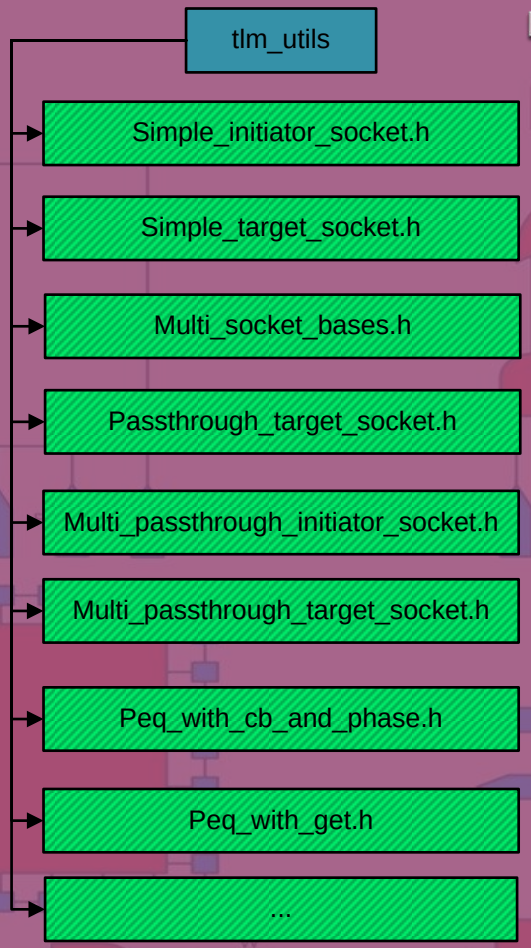


* Open SystemC Initiative, 31 May 2009

TLM-2.0 Library Hierarchy



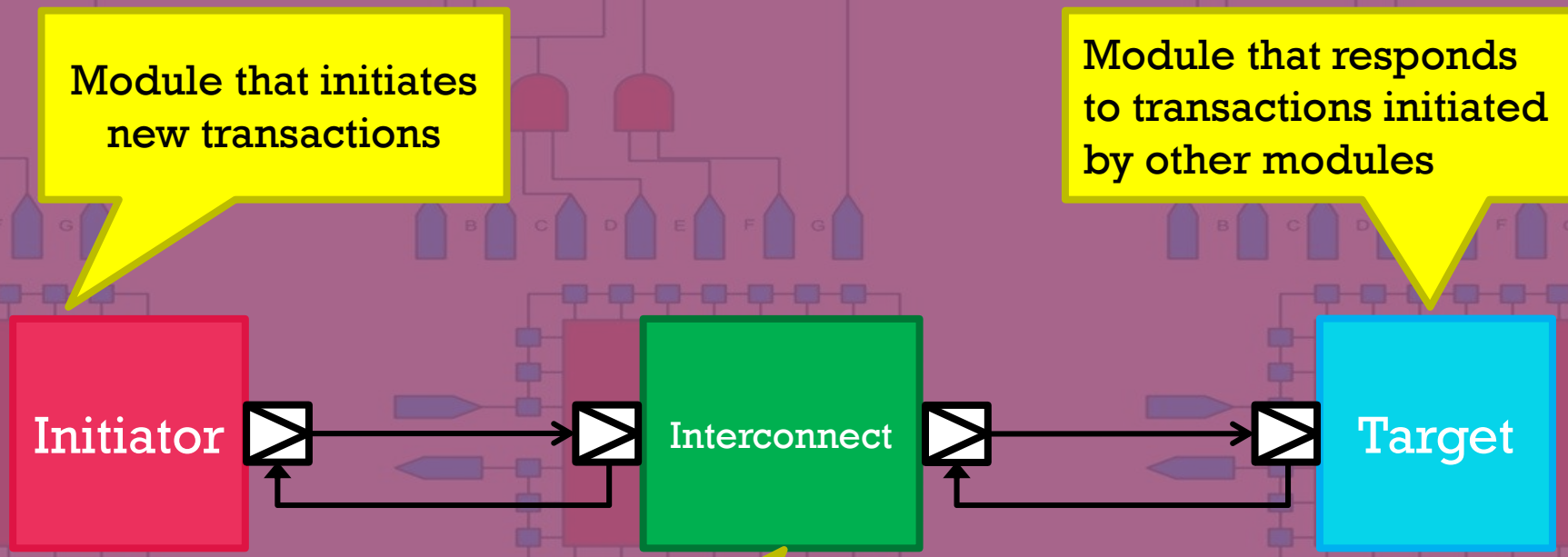




Transaction Level Modeling

- Introduction
- Processing Elements
  - Initiators
  - Targets
  - Interconnects
- + Data
- + Abstract Communications
- + Complete System

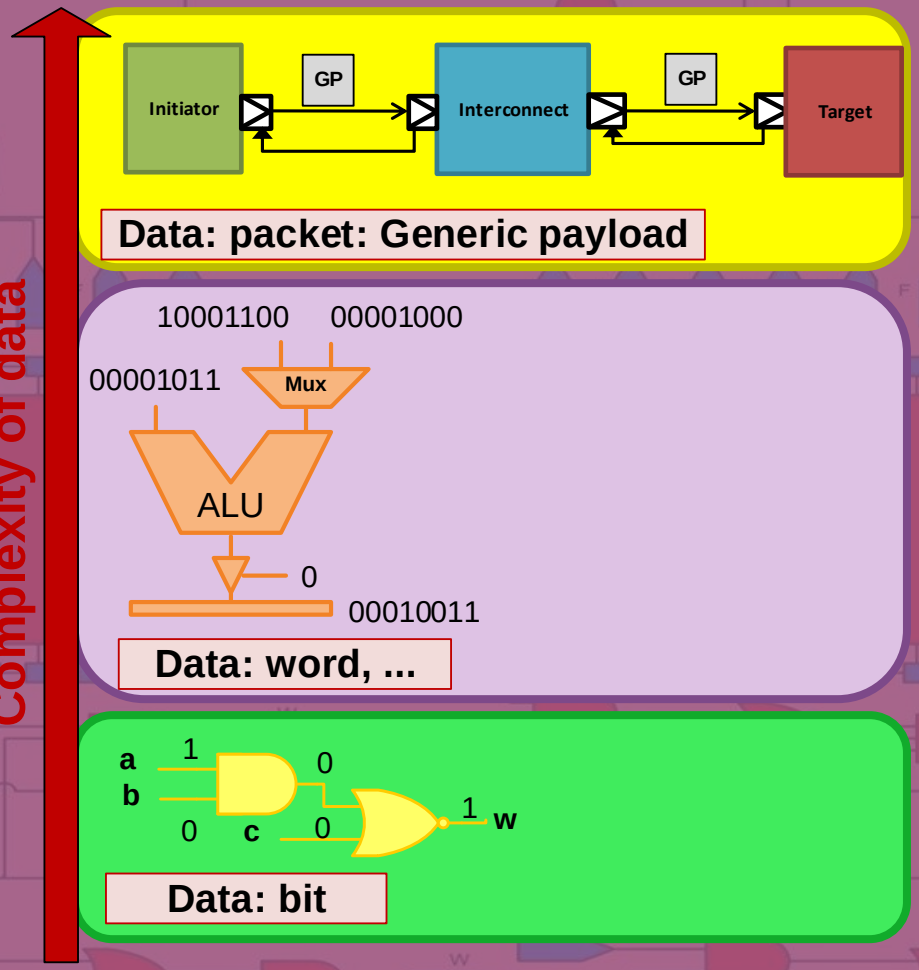
Processing Elements



Transaction Level Modeling

- Introduction
- + Processing Elements
- – Data
  - Generic Payload
- + Data
- + Abstract Communications
- + Complete System

The Generic Payload



- Standard type of the data that is transferred at system level
- Includes some of the typical memory-mapped bus protocols attributes such as:
  - Command, address, data, burst transfers, …
- Includes an extension mechanism so that applications can add their own specialized attributes

GP: *tlm_gp.h* outline

tlm_gp.h

```
namespace tlm {

class
tlm_generic_payload;

class tlm_mm_interface { ... };
//------------------------------------------------------------------------
// Classes and helper functions for the extension mechanism
//------------------------------------------------------------------------
inline unsigned int max_num_extensions(bool increment=false) { ... }
class tlm_extension_base { ... };
template <typename T>
class tlm_extension { ... };
template <typename T>
const
unsigned int tlm_extension<T>::ID = tlm_extension_base::register_extension();

//------------------------------------------------------------------------
// enumeration types
//------------------------------------------------------------------------
enum tlm_command { ... };
enum tlm_response_status { ... };
#define TLM_BYTE_DISABLED 0x0
#define TLM_BYTE_ENABLED 0xff

//------------------------------------------------------------------------
// The generic payload class:
//------------------------------------------------------------------------
class tlm_generic_payload { ... };
} // namespace tlm
```

© Zainalabedin Navabi – Transaction Level Modeling

GP: the Generic Payload Class

```
82  class tlm_generic_payload {
371 private:
372     sc_dt::uint64         m_address;
373     tlm_command           m_command;
374     unsigned char*        m_data;
375     unsigned int          m_length;
376     tlm_response_status   m_response_status;
377     bool                  m_dmi;
378     unsigned char*        m_byte_enable;
379     unsigned int          m_byte_enable_length;
380     unsigned int          m_streaming_width;
381
382 public:
383
384     /* ------------------------------------------ */
385     /* Dynamic extension mechanism:               */
386     /* ------------------------------------------ */
387     ...
502 private:
503     tlm_array<tlm_extension_base*> m_extensions;
504     tlm_mm_interface*              m_mm;
505     unsigned int                   m_ref_count;
506 };
367 ...
371 private:
```

class *tlm_generic_payload*

GP Attributes

Attribute	Type	Modifiable?
Command	tlm_command	No
Address	uint64	Interconnect only
Data pointer	unsigned char*	No (array – yes)
Data length	unsigned int	No
Byte enable pointer	unsigned char*	No (array – yes)
Byte enable length	unsigned int	No
Streaming width	unsigned int	No
DMI hint	bool	Yes
Response status	tlm_response_status	Target only
Extensions	(tlm_extension_base*)[ ]	Yes

* Open SystemC Initiative, 31 May 2009

GP Attributes, Command

- Shall be set by the **initiator**, and shall not be overwritten by any interconnect component or target

```
enum tlm_command {
    TLM_READ_COMMAND,
    TLM_WRITE_COMMAND,
    TLM_IGNORE_COMMAND
};
```

Copy from target to data array

Copy from data array to target

Neither, but may use extensions

```
tlm_command     get_command()  const ;
void            set_command( const tlm_command command ) ;
```

GP Attributes, Address

- Shall be set by the **initiator**, but may be overwritten by one or more interconnect components.

```
sc_dt::uint64      get_address()  const;
void               set_address( const sc_dt::uint64 address );
```

GP Attributes, Data Pointer

- Is a pointer to the data array
- Shall be set by the **initiator**, and shall not be overwritten by any interconnect component or target
- Methods **get_data_ptr()** and **set_data_ptr()** get or set the value of the pointer, not the contents of the array

```
unsigned char*    get_data_ptr()  const;
void              set_data_ptr( unsigned char* data );
```

GP Attributes, Data Length

- Shall be set by the **initiator**, and shall not be overwritten by any interconnect component or target
- Shall not be set to 0
  - To transfer zero bytes, the command attribute should be set to TLM_IGNORE_COMMANDs
- Data length <= BUSWIDTH / 8: single-word transfer
- Data length > BUSWIDTH / 8: burst transfer

```
unsigned int    get_data_length()  const;
void            set_data_length( const unsigned int length );
```

GP Attributes, Byte Enable Pointer

- Shall be set by the **initiator**, and shall not be overwritten by any interconnect component or target
- A value of 0
  - Indicates that corresponding byte is disabled
- A value of 0xff
  - Indicates that the corresponding byte is enabled

```
unsigned char*      get_byte_enable_ptr() const;
void                set_byte_enable_ptr( unsigned char* );
```

GP Attributes, Byte Enable Length

- Shall be set by the **initiator**, and shall not be overwritten by any interconnect component or target
- Shall be calculated using the formula

byte_enable_array_index = data_array_index % byte_enable_length

```
unsigned int    get_byte_enable_length() const;
void            set_byte_enable_length( const unsigned int );
```

GP Attributes, Streaming Width

- Determines the number of bytes transferred on each beat
- The bytes within the data array have a corresponding local addresses within the component accessing the GP
  - The lowest address: the value of the address attribute
  - The highest address:

address_attribute + streaming_width – 1

```
unsigned int    get_streaming_width() const;
void            set_streaming_width( const unsigned int );
```

GP Attributes, DMI Allowed

- Provides a hint to an initiator that it may try to obtain a direct memory pointer
- The **target** should set this attribute to true if the transaction could have been done through DMI

```
void        set_dmi_allowed( bool );
bool        is_dmi_allowed() const;
```

GP Attributes, Response Status

- Shall be set to TLM_INCOMPLETE_RESPONSE by the **initiator**
- May be overwritten by the **target**, but should not be overwritten by any interconnect component

tlm_response_status	get_response_status() const;
Void	set_response_status(const tlm_response_status);
std::string	get_response_string();
bool	is_response_ok();
bool	is_response_error();

GP Attributes, Response Status, Cont.

- The target may set the response status attribute:
  - TLM_OK_RESPONSE: Indicates that target was able to execute the command successfully
  - One of the responses listed in the following table to indicate status

enum tlm_response_status	Meaning
TLM_OK_RESPONSE	Successful
TLM_INCOMPLETE_RESPONSE	Transaction not delivered to target. (Default)
TLM_ADDRESS_ERROR_RESPONSE	Unable to act on address
TLM_COMMAND_ERROR_RESPONSE	Unable to execute command
TLM_BURST_ERROR_RESPONSE	Unable to act on data length or streaming width
TLM_BYTE_ENABLE_ERROR_RESPONSE	Unable to act on byte enable
TLM_GENERIC_ERROR_RESPONSE	Any other error

GP Extensions

- Generic payload has an array-of-pointers to extensions
- One pointer per extension type
- Every transaction can potentially carry every extension type
- Flexible mechanism

```
template <typename T>     T*  set_extension ( T* ext );
template <typename T>     T*  set_auto_extension ( T* ext );
template <typename T>     T*  get_extension() const;
template <typename T>  void   clear_extension ();
template <typename T>  void   release_extension ();
```

Freed by ref counting

Clears pointer, not extension object

mm => convert to auto
no mm => free extension object

Transaction Level Modeling

- Introduction
+ Processing Elements
+ Data
– Abstract Communications
  + Coding Styles
    - Loosely Timed
    - Approximately Timed
  + TLM-2.0 Transport Interfaces
    - Blocking Transport
      - Path (Forward & Return)
      - Socket (Simple Socket)
    - Simple Transport Example
    - Non-blocking Transport
      - Path (Forward, Backward & Return)
      - Socket (Tagged & Multi Socket)
      - Phases & Base Protocol
+ Complete System

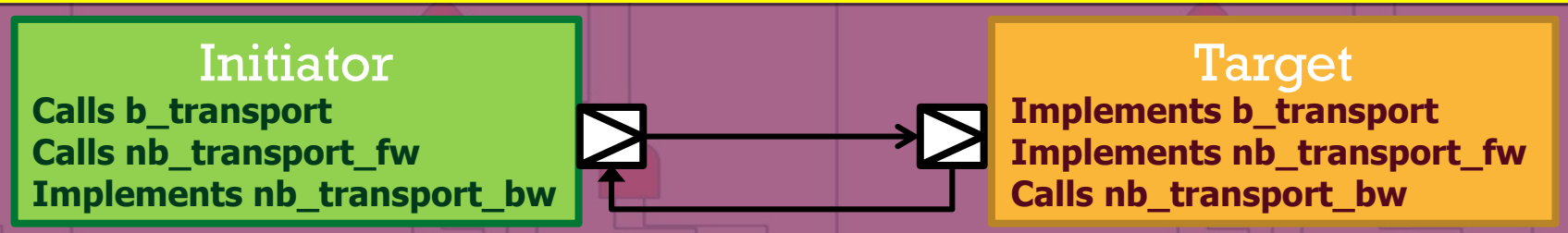
Coding Styles

- Loosely-timed
  - Each transaction has 2 timing points: *begin* and *end*
  - Uses blocking transport interface
- Approximately-timed
  - Also referred as: cycle-approximate or cycle-count-accurate
  - Sufficient for architectural exploration
  - Each transaction has 4 timing points (extensible)
  - Uses non-blocking transport interface

Transaction Level Modeling

- Introduction
- + Processing Elements
- + Data
- – Abstract Communications
  - + Coding Styles
    - Loosely Timed
    - Approximately Timed
  - + TLM-2.0 Transport Interfaces
    - Blocking Transport
      - Path (Forward & Return)
      - Socket (Simple Socket)
    - Simple Transport Example
    - Non-blocking Transport
      - Path (Forward, Backward & Return)
      - Socket (Tagged & Multi Socket)
      - Phases & Base Protocol
  - + Complete System

Transport Interfaces



tlm_blocking_transport_if

```
void b_transport( TRANS& , sc_time& ) ;
```

tlm_fw_nonblocking_transport_if

```
tlm_sync_enum  nb_transport_fw( TRANS& , PHASE& , sc_time& );
```

returns a value indicating whether or not the return path was used

tlm_bw_nonblocking_transport_if

```
tlm_sync_enum  nb_transport_bw( TRANS& , PHASE& , sc_time& );
```

Blocking Transport

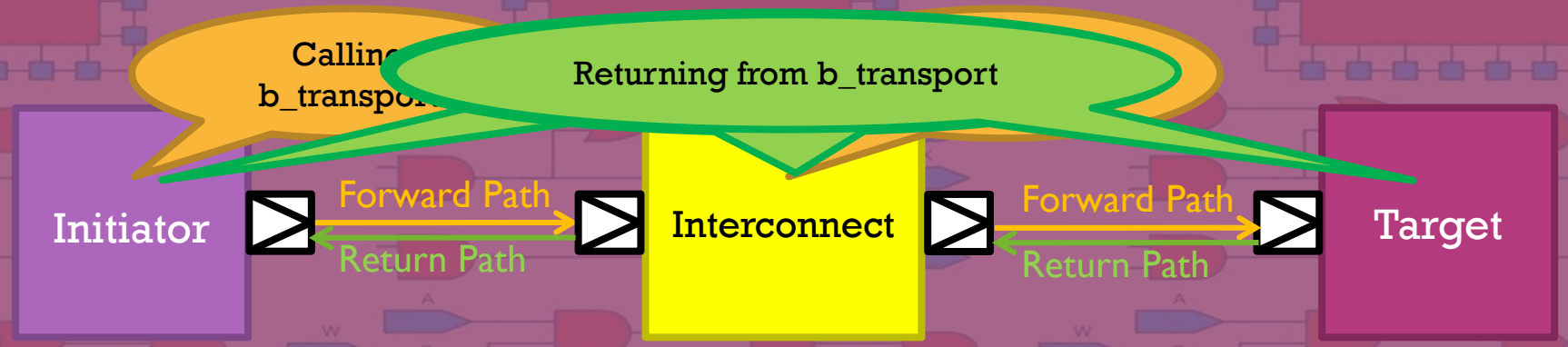
- Blocking transport interface
  - Includes timing annotation
  - Typically used with loosely-timed coding style
  - Forward path only

```
template < typename  TRANS  =  tlm_generic_payload >

class tlm_blocking_transport_if : public virtual sc_core::sc_interface {
public:
      virtual  void  b_transport ( TRANS& trans , sc_core::sc_time& t )  =  0;
};
```

Blocking Transport: Path

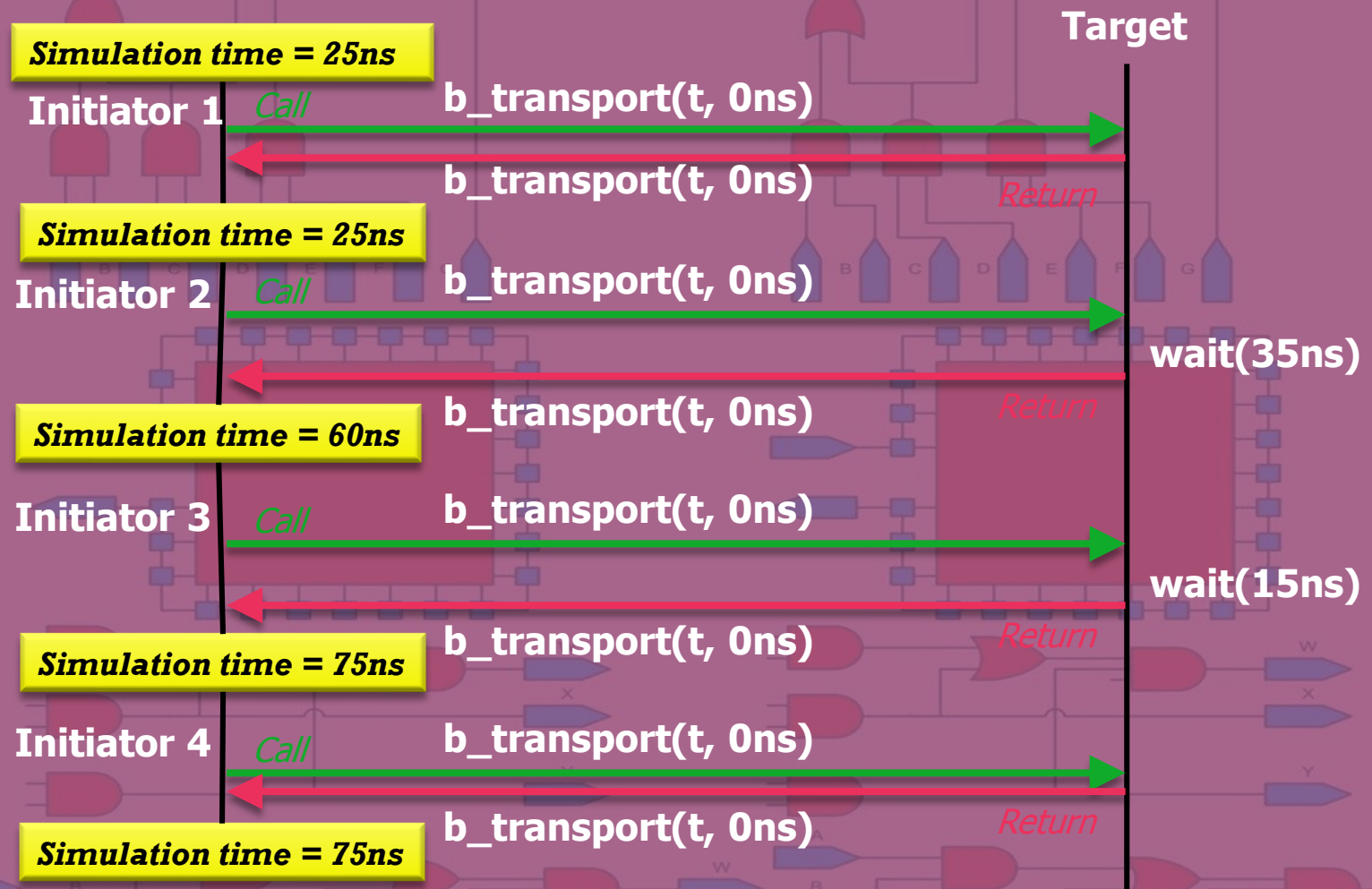
- Forward path
  - Is the calling path by which an initiator or interconnect component makes interface method calls forward in the direction of another interconnect component or the target
- Return path
  - Is the return path by which an initiator, target or interconnect component returns immediately from the interface method that has been called



Blocking Transport: Path

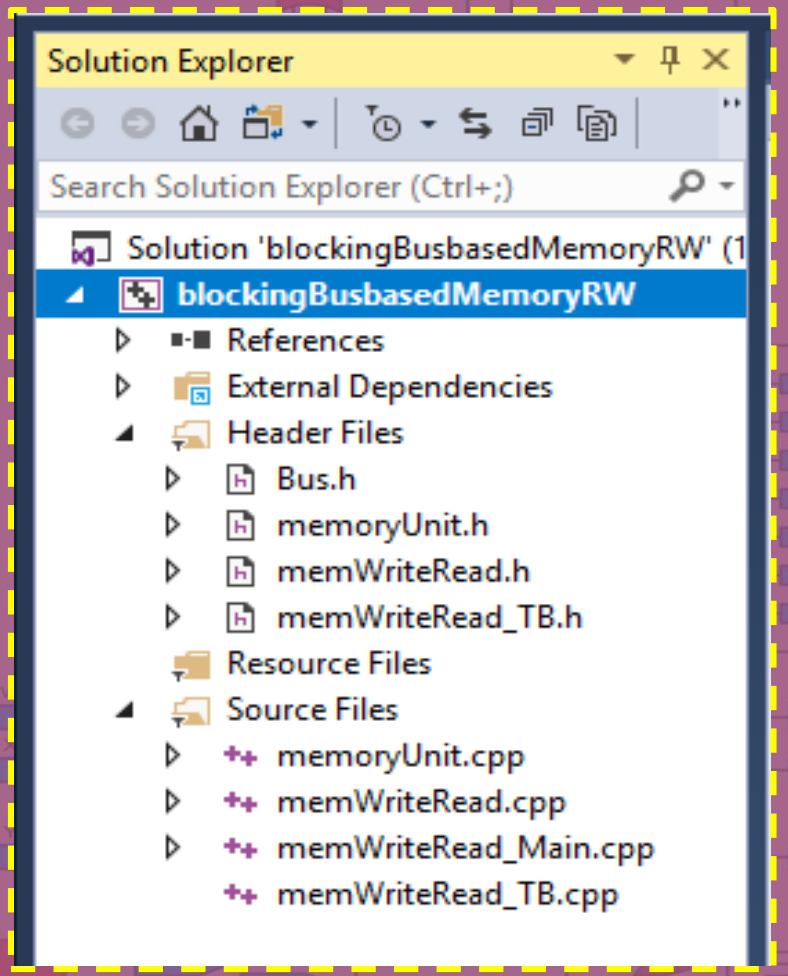
- Timing Diagram

Initiator is blocked until return from b_transport



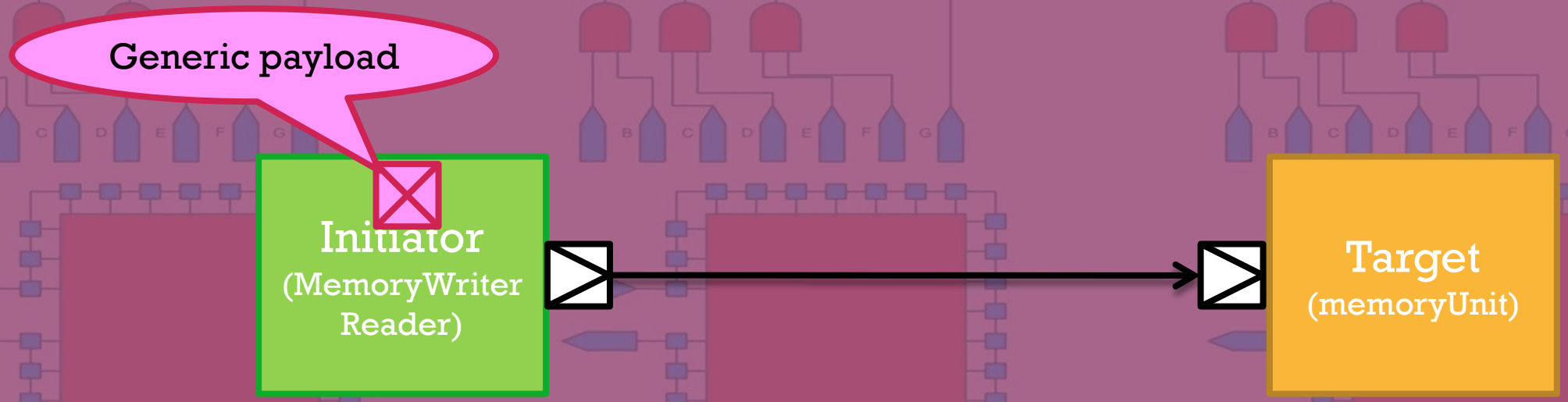
Transaction Level Modeling

- Introduction
- + Processing Elements
- + Data
- – Abstract Communications
  - + Coding Styles
  - + TLM-2.0 Transport Interfaces
    - Blocking Transport
      - Path (Forward & Return)
      - blockingBusbasedMemoryRW Example
      - Socket (Simple Socket)
    - Simple Transport Example
    - Non-blocking Transport
- + Complete System



Blocking Transport: Path

- Example 1: blockingBusbasedMemoryRW



Blocking Transport: Path

- Example 1: blockingBusbasedMemoryRW: Initiator, *memWriterReader*

memWriterReader.h

memWriteRead.h | memWriteRead_TB.h | memoryUnit.cpp* | memWriteRead.cpp

blockingBusbasedMemoryRW | (Global Scope)

```
1   #include <systemc.h>
2   #include "tlm.h"
3
4   class memWriterReader : public sc_module {
5   public:
6       sc_port<tlm::tlm_blocking_transport_if<>> memWriterReaderBus;
7
8       SC_CTOR(memWriterReader) :blockWriteRead(0) {
9           blockWriteRead = new tlm::tlm_generic_payload;
10
11          for (int i = 0; i < 4; i++) *(data+i) = i+192;
12
13          SC_THREAD(memWritingReading);
14          }
15
16      void memWritingReading();
17      tlm::tlm_generic_payload* blockWriteRead;
18
19      sc_lv<8> data[5];
20  };
```

Blocking Transport: Path

- Example 1: blockingBusbasedMemoryRW: Initiator, *memWriterReader*

memWriterReader.cpp

```
#include "memWriteRead.h"

void memWriterReader::memWritingReading(){
    sc_time blockedTime = sc_time(13, SC_NS);
    sc_time pauseTime = sc_time(15, SC_NS);

    for (int i = 0; i < 111; i += 11) {
        tlm::tlm_command cmd = (tlm::tlm_command)(rand() % 2);
        if (cmd == tlm::TLM_WRITE_COMMAND) {
            data[0] = (sc_lv<8>) (i+5);
            data[1] = (sc_lv<8>) (i+6);
            data[2] = (sc_lv<8>) (i+7);
            data[3] = (sc_lv<8>) (i+8);
            data[4] = (sc_lv<8>) (i+9);
        }

        blockWriteRead->set_command( cmd );
        blockWriteRead->set_address( i );
        blockWriteRead->set_data_ptr( (unsigned char*) data );
        blockWriteRead->set_data_length( 5 );
        blockWriteRead->set_streaming_width( 5 );
        blockWriteRead->set_byte_enable_ptr( 0 );
        blockWriteRead->set_dmi_allowed( false );
        blockWriteRead->set_response_status( tlm::TLM_INCOMPLETE_RESPONSE );

        memWriterReaderBus->b_transport( *blockWriteRead, blockedTime );

        cout << "WR: " << (cmd ? 'W' : 'R') << ", @" << i << " data:";
        sc_lv<8> vv;
        for(int j=0; j<5; j++) {vv=data[j]; cout << vv << " ";} cout << '\n';

        wait(pauseTime);
    }
}
```

Set GP parameters

In terms of bytes

= data_length to indicate no streaming

initial value

Calling blocking transport

Blocking Transport: Path

- Example 1: blockingBusbasedMemoryRW: Target, *memoryUnit*

memoryUnit.h

memWriteRead_TB.cpp | memWriteRead_Main.cpp | Bus.h | memoryUnit.h

blockingBusbasedMemoryRW | (Global Scope)

```
1   #include <systemc.h>
2   #include "tlm.h"
3   #include "Bus.h"
4
5   SC_MODULE (memoryUnit) {
6       sc_export<tlm::tlm_blocking_transport_if<>> memSideBus;
7       bus *membus;
8       static const int SIZE=256;
9
10      SC_CTOR(memoryUnit) : memSideBus("memorySideBus")
11          {
12              membus = new bus();
13              memSideBus(*membus); // Can go in the TB
14              for (int i = 0; i < SIZE; i++)
15                  memArray[i] = (sc_lv<8>) (i%256 + 192);
16      }
17  public:
18      virtual void b_transport( tlm::tlm_generic_payload&, sc_time&);
19      sc_lv<8> memArray[SIZE];
20  };
21
```

Blocking Transport: Path

memWriteRead.h | memWriteRead_TB.h | memoryUnit.cpp* | memWriteRead.cpp

blockingBusbasedMemoryRW | (Global Scope)

```
1   #include "memoryUnit.h"
2
3   void memoryUnit::b_transport( tlm::tlm_generic_payload& gotThis,
4                                 sc_time& delayValue )
5   {
6       tlm::tlm_command cmd = gotThis.get_command();
7       uint64           adr = gotThis.get_address();
8       unsigned char*   ptr = gotThis.get_data_ptr();
9       unsigned int     len = gotThis.get_data_length();
10      unsigned char*   byt = gotThis.get_byte_enable_ptr();
11      unsigned int     wid = gotThis.get_streaming_width();
12
13      if (adr >= uint64(SIZE) || byt != 0 || len > 5 || wid < len)
14          SC_REPORT_ERROR("TLM-2.0: ","Inconsistant Generic Payload");
15
16      unsigned int i;
17      if ( cmd == tlm::TLM_READ_COMMAND ){
18          for(i=0; i<len; i=i+1) {
19              *(ptr+i) = *((unsigned char*) (memArray+adr+i));
20          }
21      }
22      else if ( cmd == tlm::TLM_WRITE_COMMAND ){
23          for(i=0; i<len; i=i+1) {
24              *((unsigned char*) (memArray+adr+i)) = *(ptr+i);
25          }
26      }
27
28      gotThis.set_response_status( tlm::TLM_OK_RESPONSE );
29      wait(delayValue);
30  }
```

memoryUnit.cpp

- Example 1: blockingBusbasedMemoryRW: Target, *memoryUnit*

Obliged to check address range and check for unsupported features, using the SystemC report handler is an acceptable way of signalling an error

Obliged to implement read and write commands

Obliged to set response status to indicate successful completion

Blocking Transport: Path

- Example 1: blockingBusbasedMemoryRW: Interface Defenition & Implementation , *Bus*

Bus.h

Bus.h* memoryUnit.h

blockingBusbasedMemoryRW (Global Scope)

```
1   #include <systemc.h>
2   #include "tlm.h"
3
4   class bus : public tlm::tlm_blocking_transport_if<>
5   {
6   public:
7       bus() {};
8       ~bus() {};
9
10      virtual void b_transport(tlm::tlm_generic_payload&, sc_time&){};
11  };
```

Blocking Transport: Path

- Example 1: blockingBusbasedMemoryRW: Testbench

memWriteRead_TB.cpp | Bus.h | memoryUnit.h | memWriteRead.h | memWriteRead_TB.h

blockingBusbasedMemoryRW | (Global Scope)

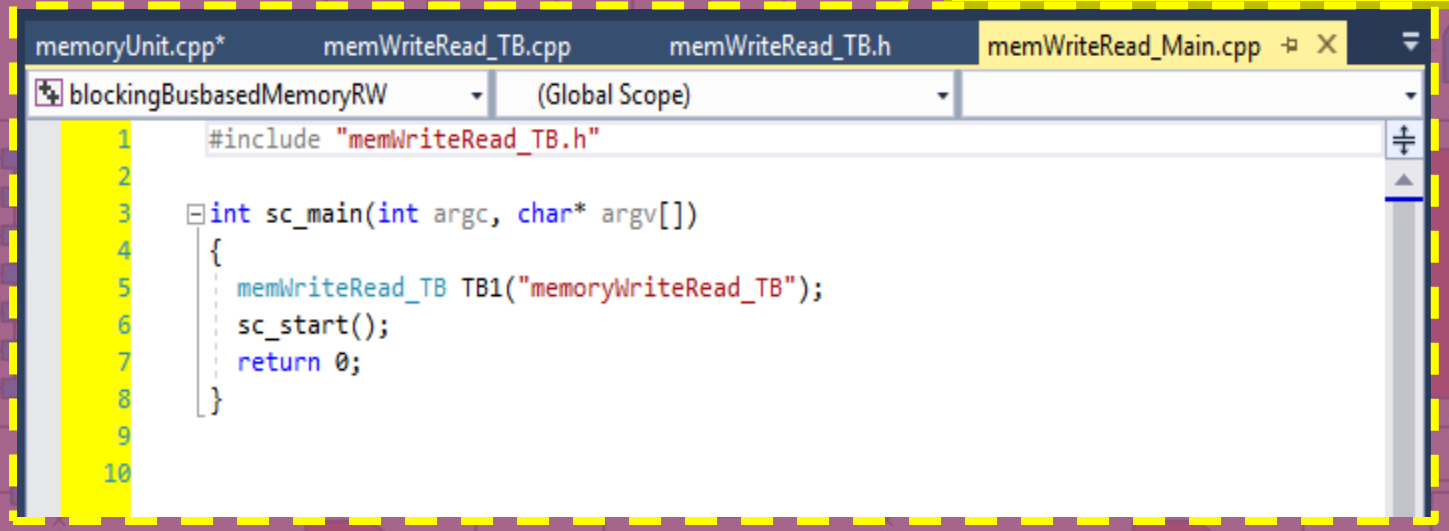
```
1   #include "memWriteRead.h"
2   #include "memoryUnit.h"
3
4   SC_MODULE (memWriteRead_TB) {
5       memWriterReader* MW1;
6       memoryUnit* MU1;
7
8       SC_CTOR(memWriteRead_TB)
9           {
10          MU1 = new memoryUnit("memoryUnit");
11              // MU1->memSideBus(*MU1->membus); // This is in memoryUnit
12          MW1 = new memWriterReader("memoryWriter");
13              MW1->memWriterReaderBus(*MU1->membus);
14      }
15  };
16
```

memoryWrite_TB.h

Blocking Transport: Path

- Example 1: blockingBusbasedMemoryRW: Main

memWriteRead_Main.cpp



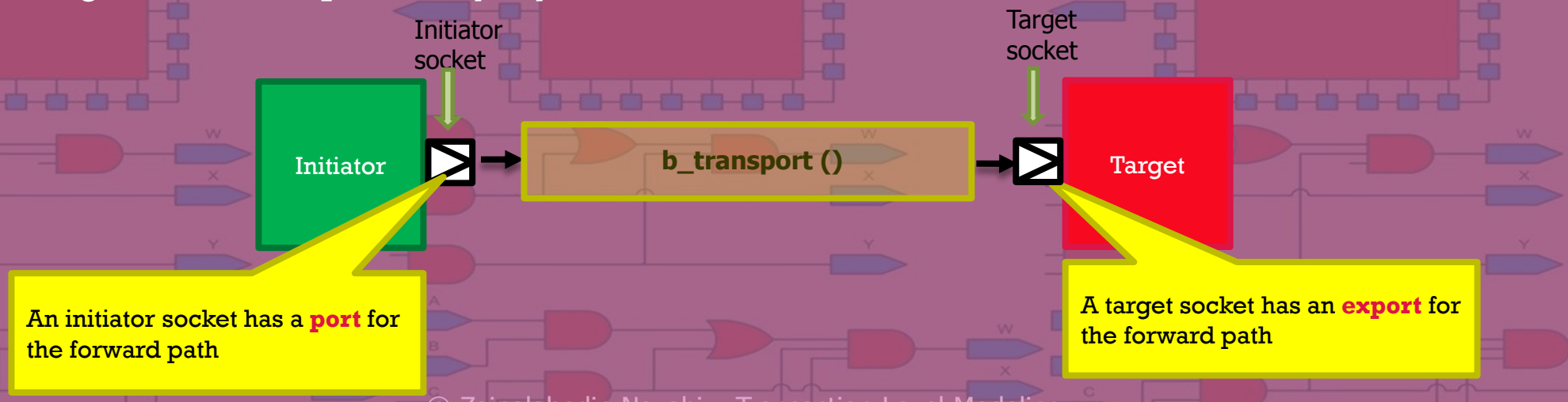
Blocking Transport: Path

- Example 1: blockingBusbasedMemoryRW: Output

```
        SystemC 2.3.2-Accellera --- Nov 16 2018 05:36:55
        Copyright (c) 1996-2017 by all Contributors,
        ALL RIGHTS RESERVED
WR: W, @0 data:00000101 00000110 00000111 00001000 00001001
WR: W, @11 data:00010000 00010001 00010010 00010011 00010100
WR: R, @22 data:00010000 00010001 00010010 00010011 00010100
WR: R, @33 data:00010000 00010001 00010010 00010011 00010100
WR: W, @44 data:00110001 00110010 00110011 00110100 00110101
WR: R, @55 data:00110001 00110010 00110011 00110100 00110101
WR: R, @66 data:00110001 00110010 00110011 00110100 00110101
WR: R, @77 data:00110001 00110010 00110011 00110100 00110101
WR: R, @88 data:00110001 00110010 00110011 00110100 00110101
WR: R, @99 data:00110001 00110010 00110011 00110100 00110101
WR: W, @110 data:01110011 01110100 01110101 01110110 01110111
```

Blocking Transport: Socket (Simple)

- Combine a port with an export
- Provide methods to bind port and export of the paths
- Offer strong type checking when binding sockets parameterized with incompatible protocol types
- The classes **tlm_initiator_socket** and **tlm_target_socket** are typically used directly by applications
  - Belong to the interoperability layer of the TLM-2.0 standard



Blocking Transport: Socket (Concepts)

tlm_initiator_socket.h

```
namespace tlm {
template <unsigned int BUSWIDTH = 32,
          typename FW_IF = tlm_fw_transport_if<>,
          typename BW_IF = tlm_bw_transport_if<> >
class tlm_base_initiator_socket_b
{
public:
  virtual ~tlm_base_initiator_socket_b() { ... }
  virtual sc_core::sc_port_b<FW_IF> & get_base_port() = 0;
  virtual                     BW_IF  & get_base_interface() = 0;
  virtual sc_core::sc_export<BW_IF> & get_base_export() = 0;
};

template <unsigned int BUSWIDTH,
          typename FW_IF,
          typename BW_IF> class tlm_base_target_socket_b;

template <unsigned int BUSWIDTH,
          typename FW_IF,
          typename BW_IF,
          int N...> class tlm_base_target_socket;

template <unsigned int BUSWIDTH = 32,
          typename FW_IF = tlm_fw_transport_if<>,
          typename BW_IF = tlm_bw_transport_if<>,
          int N = 1...>
class tlm_base_initiator_socket { ... };
/* ... */

template <unsigned int BUSWIDTH = 32,
          typename TYPES = tlm_base_protocol_types,
          int N = 1... >
class tlm_initiator_socket { ... };
} // namespace tlm
```

- Class Definition

Port and export

Blocking Transport: Socket (Concepts)

```
59  class tlm_base_initiator_socket : public tlm_base_initiator_socket_b<BUSWIDTH, FW_IF, BW_IF>,
60                                    public sc_core::sc_port<FW_IF, N... >
65  {
66  public:
67    typedef FW_IF                                  fw_interface_type;
68    typedef BW_IF                                  bw_interface_type;
69    typedef sc_core::sc_port<fw_interface_type, N...>   port_type;
74    typedef sc_core::sc_export<bw_interface_type> export_type;
75    typedef tlm_base_target_socket_b<BUSWIDTH,
76                                     fw_interface_type,
77                                     bw_interface_type> base_target_...ype;
78    typedef tlm_base_initiator_socket_b<BUSWIDTH,
79                                        fw_interface_type,
80                                        bw_interface_type> base_type;
81
82    template <unsigned int, typename, typename, int... >
87    friend class tlm_base_target_socket;
88  public:
89    tlm_base_initiator_socket() { ... }
95    explicit tlm_base_initiator_socket(const char* name) { ... }
101   virtual const char* kind() const { ... }
106   unsigned int get_bus_width() const...
118   void bind(base_target_socket_type& s) { ... }
126   void operator() (base_target_socket_type& s)...
135   void bind(base_type& s) { ... }
143   void operator() (base_type& s)...
152   void bind(bw_interface_type& ifs) { ... }
157   void operator() (bw_interface_type& s) { ... }
162   // Implementation of pure virtual functions of base class
163   virtual sc_core::sc_port_b<FW_IF> &   get_base_port()      { return *this;   }
164   virtual                     BW_IF  &   get_base_interface() { return m_export; }
165   virtual sc_core::sc_export<BW_IF> &   get_base_export() { ... }
167 protected:
168   export_type m_export;
169 };
```

tlm_initiator_socket.h: base class

⊙ Class Definition, Cont.

Defining fw and bw interfaces

Defining port and export

Binding and overloading of ()

Base class for initiator sockets, providing binding methods

Blocking Transport: Socket (Concepts)

⊙ Class Definition, Cont.

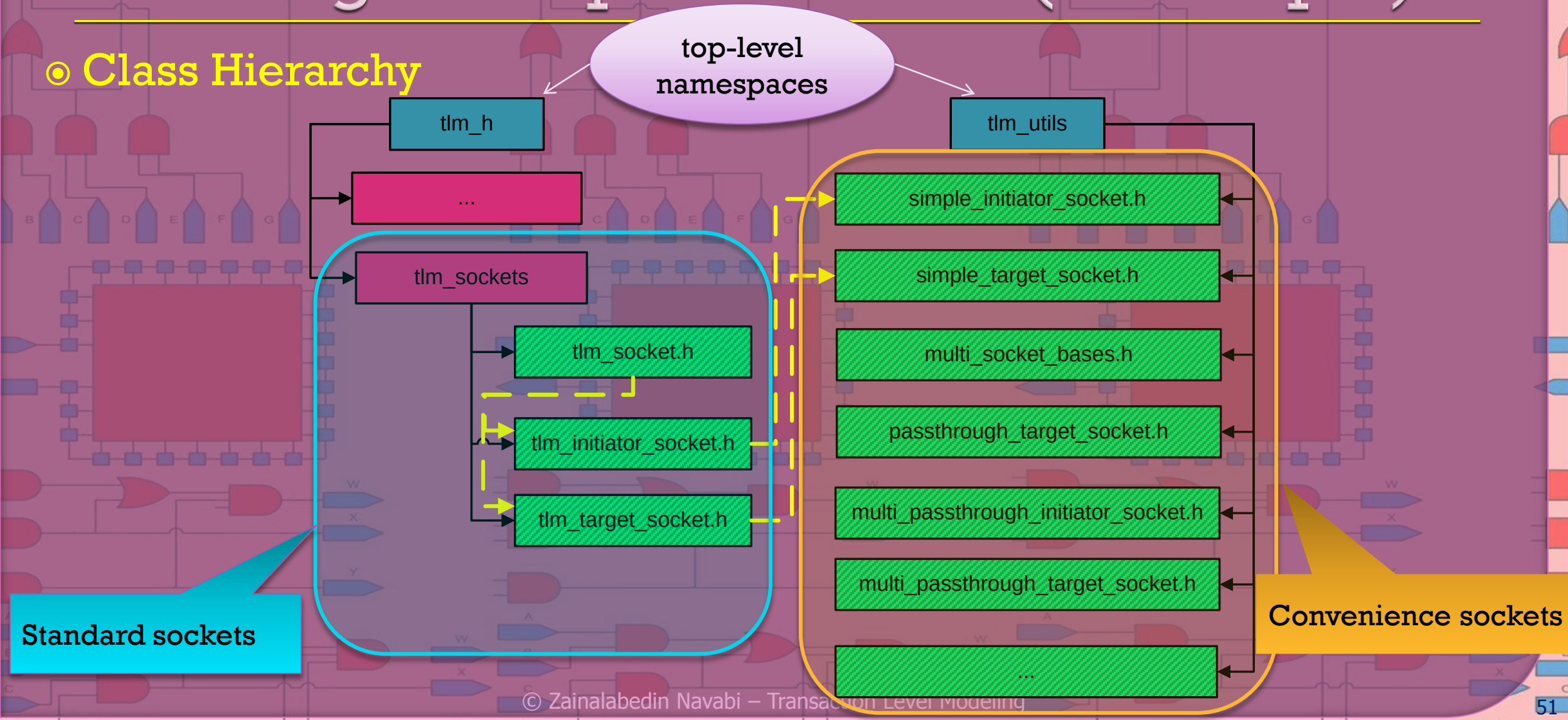
tlm_initiator_socket.h: the conveninence class

Is inherited from the *tlm_initiator_socket_base* class

```
182  class tlm_initiator_socket :
       public tlm_base_initiator_socket <BUSWIDTH,
                                         tlm_fw_transport_if<TYPES>,
185                                      tlm_bw_transport_if<TYPES>,
186                                      N...>
1    {
2    public:
3      tlm_initiator_socket() :
4        tlm_base_initiator_socket<BUSWIDTH,
5                                  tlm_fw_transport_if<TYPES>,
196                                tlm_bw_transport_if<TYPES>,
197                                N... >() { ... }
205    explicit tlm_initiator_socket(const char* name) :
206      tlm_base_initiator_socket<BUSWIDTH,
207                                tlm_fw_transport_if<TYPES>,
208                                tlm_bw_transport_if<TYPES>,
209                                N... >(name)
214    {
215    }
216
217    virtual const char* kind() const { ... }
221  };
```

Blocking Transport: Socket (Concepts)

- Class Hierarchy



Blocking Transport: Socket (Simple)

- **Simple Sockets**
  - Simple to use
  - Derived from the interoperability layer sockets **tlm_initiator_socket** and **tlm_target_socket**
  - Provide methods for registering callback methods instead of binding sockets to objects

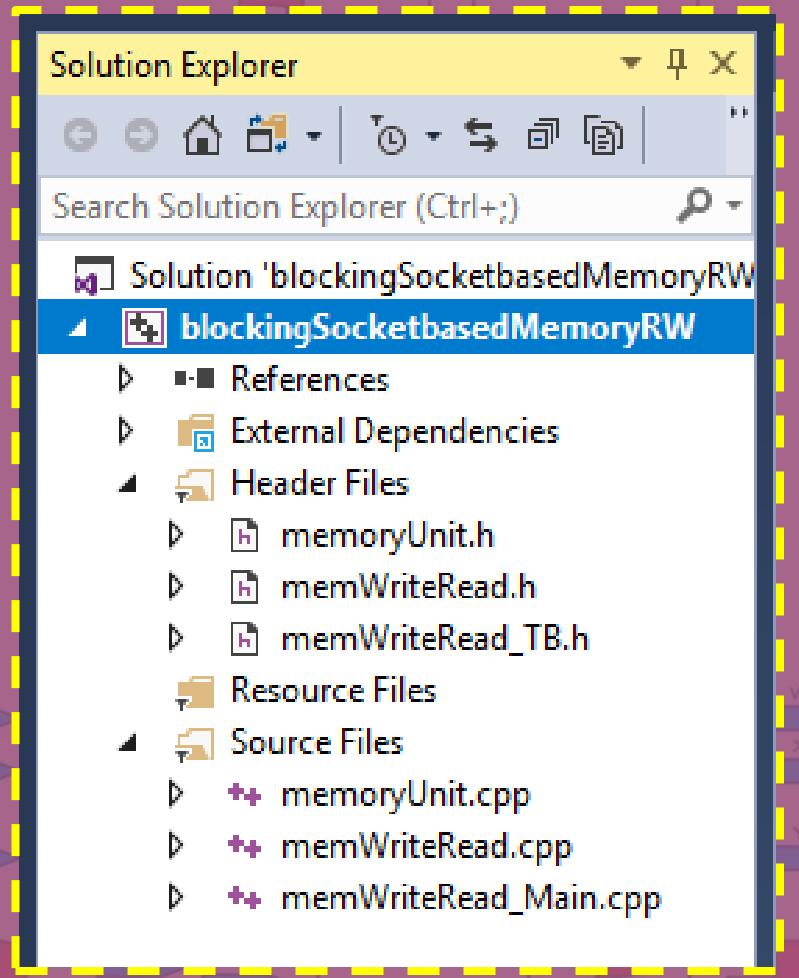
Blocking Transport: Socket (Simple)

```
70  class process : public tlm::tlm_bw_transport_if<TYPES>
71  {
72  public:
73    typedef sync_enum_type (MODULE::*TransportPtr)(transaction_type&,
74                                                   phase_type&,
75                                                   sc_core::sc_time&);
76    typedef void (MODULE::*InvalidateDirectMemPtr)(sc_dt::uint64,
77                                                   sc_dt::uint64);
78
79    process(const std::string& name) { ... }
86
87    void set_transport_ptr(MODULE* mod, TransportPtr p) { ... }
100   void set_invalidate_direct_mem_ptr(MODULE* mod, InvalidateDirectMemPtr p) { ... }
113   sync_enum_type nb_transport_bw(transaction_type& trans, phase_type& phase, sc_core::sc_time& t) { ... }
128   void invalidate_direct_mem_ptr(sc_dt::uint64 start_range,
129                                  sc_dt::uint64 end_range) { ... }
138   private:
139     sync_enum_type nb_transport_bw(transaction_type& trans, phase_type& phase, sc_core::sc_time& t)
140     {
141       if (m_transport_ptr) {
142         // forward call
143         assert(m_mod);
144         return (m_mod->*m_transport_ptr)(trans, phase, t);
145
146       } else {
147         std::stringstream s;
148         s << m_name << ": no transport callback registered";
149         SC_REPORT_ERROR("/OSCI_TLM-2/simple_socket",s.str().c_str());
150       }
151       return tlm::TLM_ACCEPTED;    ///< unreachable code
152     }
153
154 };
```

- Simple Initiator Socket: Class Definition

Transaction Level Modeling

- Introduction
- + Processing Elements
- + Data
- – Abstract Communications
  - + Coding Styles
  - + TLM-2.0 Transport Interfaces
    - Blocking Transport
      - Path (Forward & Return)
      - Socket (Simple Socket)
      - blockingSocketbasedMemoryRW Example
    - Simple Transport Example
    - Non-blocking Transport
- + Complete System



Blocking Transport: Socket (Simple)

- Example 2: blockingSocketbasedMemoryRW: Initiator, *memWriterReader*

memWriterReader.h

memWriteRead_Main.cpp | memWriteRead.cpp | memoryUnit.h | memWriteRead.h

blockingSocketbasedMemoryRW | (Global Scope)

```
1  #include <systemc.h>
2  #include "tlm.h"
3  #include "tlm_utils/simple_target_socket.h"
4  #include "tlm_utils/simple_initiator_socket.h"
5
6  class memWriterReader : public sc_module {
7  public:
8      tlm_utils::simple_initiator_socket<memWriterReader> memWriterReaderSocket;
9
10     SC_CTOR(memWriterReader) : memWriterReaderSocket("Writer-Socket"),
11                                blockWriteRead(0) {
12         blockWriteRead = new tlm::tlm_generic_payload;
13
14         for (int i = 0; i < 4; i++) *(data+i) = i+192;
15
16         SC_THREAD(memWritingReading);
17         }
18
19     void memWritingReading();
20     tlm::tlm_generic_payload* blockWriteRead;
21
22     sc_lv<8> data[5];
23 };
```

Defining the Initiator socket and specializing it for memWriterReader

Initializing memory

Blocking Transport: Socket (Simple)

memWriterReader.cpp

- Example 2: blockingSocketbasedMemoryRW: Initiator, *memWriterReader*

memWriteRead_Main.cpp | memWriteRead.cpp | memoryUnit.h | memWriteRead.h

blockingSocketbasedMemoryRW | (Global Scope)

```
1   #include "memWriteRead.h"
2
3   void memWriterReader::memWritingReading(){
4       sc_time blockedTime = sc_time(13, SC_NS);
5       sc_time pauseTime = sc_time(15, SC_NS);
6
7       for (int i = 0; i < 111; i += 11) {
8           tlm::tlm_command cmd = (tlm::tlm_command)(rand() % 2);
9           if (cmd == tlm::TLM_WRITE_COMMAND) {
10              data[0] = (sc_lv<8>) (i+5);
11              data[1] = (sc_lv<8>) (i+6);
12              data[2] = (sc_lv<8>) (i+7);
13              data[3] = (sc_lv<8>) (i+8);
14              data[4] = (sc_lv<8>) (i+9);
15          }
16
17      blockWriteRead->set_command( cmd );
18      blockWriteRead->set_address( i );
19      blockWriteRead->set_data_ptr( (unsigned char*) data );
20      blockWriteRead->set_data_length( 5 );
21      blockWriteRead->set_streaming_width( 5 );
22      blockWriteRead->set_byte_enable_ptr( 0 );
23      blockWriteRead->set_dmi_allowed( false );
24      blockWriteRead->set_response_status( tlm::TLM_INCOMPLETE_RESPONSE );
25
26      memWriterReaderSocket->b_transport( *blockWriteRead, blockedTime );
27
28      if ( blockWriteRead->is_response_error() )
29          SC_REPORT_ERROR("TLM-2", "Error in memory handling of b_transport");
30
31          cout << " At time: " << sc_time_stamp() << "WR: "
32               << (cmd ? 'W' : 'R') << ", Iteration:" << i << " data:";
33          sc_lv<8> vv;
34          for(int j=0; j<5; j++) {vv=data[j]; cout << vv << " ";} cout << '\n';
35          // wait(pauseTime);
36      }
37  }
```

Set GP parameters

= data_length to indicate no streaming

Mandatory initial value

Initiator obliged to check response status

Blocking Transport: Socket (Simple)

- Example 2: blockingSocketbasedMemoryRW: Target, *memoryUnit*

memoryUnit.h

memWriteRead_Main.cpp | memWriteRead.cpp | memoryUnit.h | memWri

blockingSocketbasedMemoryRW | (Global Scope)

```
1  #include <systemc.h>
2  #include "tlm.h"
3  #include "tlm_utils/simple_target_socket.h"
4  #include "tlm_utils/simple_initiator_socket.h"
5
6  SC_MODULE (memoryUnit) {
7      tlm_utils::simple_target_socket<memoryUnit> memSideSocket;
8
9      static const int SIZE=256;
10
11     SC_CTOR(memoryUnit) : memSideSocket("memorySideSocket")
12         {
13             memSideSocket.register_b_transport(this, &memoryUnit::b_transport);
14
15             for (int i = 0; i < SIZE; i++)
16                 memArray[i] = (sc_lv<8>) (i%256 + 192);
17         }
18 public:
19     virtual void b_transport( tlm::tlm_generic_payload&, sc_time&);
20     sc_lv<8> memArray[SIZE];
21 };
```

Defining the target socket and specializing it for memoryUnit

Register callback for incoming b_transport interface method call

Initializing memory

Blocking Transport: Socket (Simple)

memoryUnit.cpp

memWriteRead.cpp | memoryUnit.h | memWriteRead.h | memoryUnit.cpp

blockingSocketbasedMemoryRW | (Global Scope)

```
1   #include "memoryUnit.h"
2
3   void memoryUnit::b_transport( tlm::tlm_generic_payload& gotThis,
4                                 sc_time& delayValue )
5   {
6       tlm::tlm_command cmd = gotThis.get_command();
7       uint64           adr = gotThis.get_address();
8       unsigned char*   ptr = gotThis.get_data_ptr();
9       unsigned int     len = gotThis.get_data_length();
10      unsigned char*   byt = gotThis.get_byte_enable_ptr();
11      unsigned int     wid = gotThis.get_streaming_width();
12
13      if (adr >= uint64(SIZE) || byt != 0 || len > 5 || wid < len)
14          SC_REPORT_ERROR("TLM-2.0: ","Inconsistant Generic Payload");
15
16      unsigned int i;
17
18      if ( cmd == tlm::TLM_READ_COMMAND ){
19          for(i=0; i<len; i=i+1) {
20              *(ptr+i) = *((unsigned char*) (memArray+adr+i));
21          }
22      }
23      else if ( cmd == tlm::TLM_WRITE_COMMAND ){
24          for(i=0; i<len; i=i+1) {
25              *((unsigned char*) (memArray+adr+i)) = *(ptr+i);
26          }
27      }
28
29      gotThis.set_response_status( tlm::TLM_OK_RESPONSE );
30      wait(delayValue);
31  }
```

Obliged to implement read and write commands

Obliged to check address range and check for unsupported features, using the SystemC report handler is an acceptable way of signalling an error

Obliged to set response status to indicate successful completion

- Example 2: blockingSocketbasedMemoryRW: Target, *memoryUnit*

Blocking Transport: Socket (Simple)

- Example 2: blockingSocketbasedMemoryRW: Testbench

memWriteRead.cpp | memoryUnit.h | memoryUnit.cpp | memWriteRead_TB.h

blockingSocketbasedMemoryRW | (Global Scope)

```
1  #include "memWriteRead.h"
2  #include "memoryUnit.h"
3
4  SC_MODULE (memWriteRead_TB) {
5      memWriterReader* MW1;
6      memoryUnit* MU1;
7
8      SC_CTOR(memWriteRead_TB)
9          {
10             MW1 = new memWriterReader("memoryWriter");
11             MU1 = new memoryUnit("memoryUnit");
12             MW1->memWriterReaderSocket.bind( MU1->memSideSocket );
13         }
14 };
```

memoryWrite_TB.h

Instantiate components

- One initiator is bound directly to one target with no intervening bus
- Bind initiator socket to target socket

Blocking Transport: Socket (Simple)

- Example 2: blockingSocketbasedMemoryRW: Main

memWriteRead_Main.cpp

Blocking Transport: Socket (Simple)

- Example 2: blockingSocketbasedMemoryRW: Output

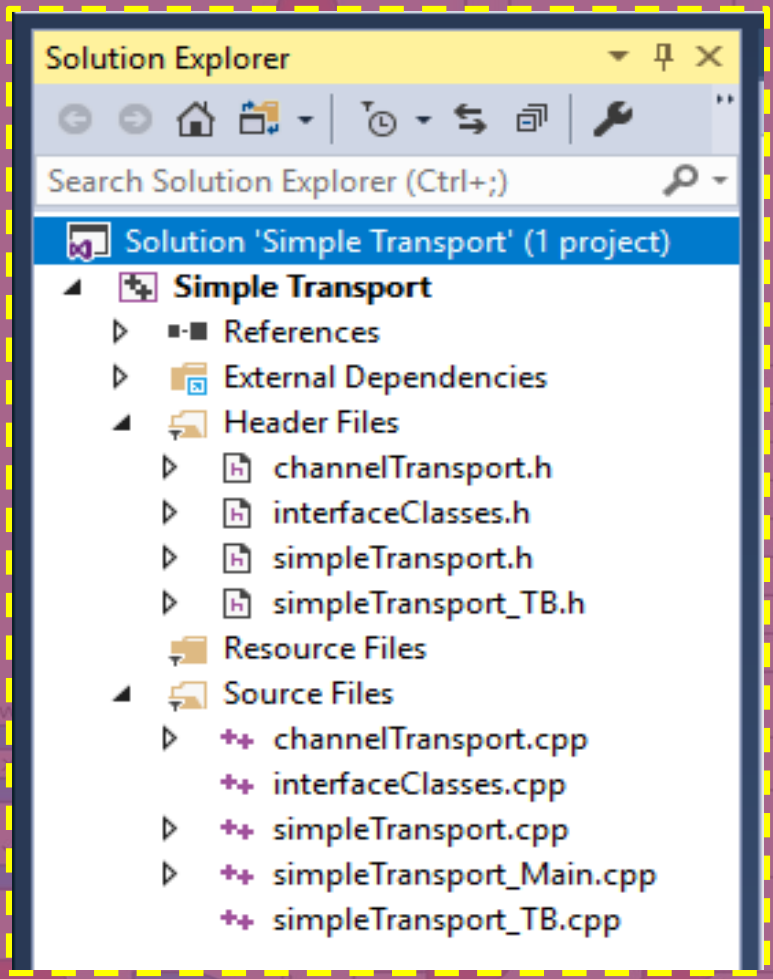
```
        SystemC 2.3.2-Accellera --- Nov 16 2018 05:36:55
        Copyright (c) 1996-2017 by all Contributors,
        ALL RIGHTS RESERVED
At time: 13 nsWR: W, Iteration:0 data:00000101 00000110 00000111 00001000 00001001
At time: 26 nsWR: W, Iteration:11 data:00010000 00010001 00010010 00010011 00010100
At time: 39 nsWR: R, Iteration:22 data:00010000 00010001 00010010 00010011 00010100
At time: 52 nsWR: R, Iteration:33 data:00010000 00010001 00010010 00010011 00010100
At time: 65 nsWR: W, Iteration:44 data:00110001 00110010 00110011 00110100 00110101
At time: 78 nsWR: R, Iteration:55 data:00110001 00110010 00110011 00110100 00110101
At time: 91 nsWR: R, Iteration:66 data:00110001 00110010 00110011 00110100 00110101
At time: 104 nsWR: R, Iteration:77 data:00110001 00110010 00110011 00110100 00110101
At time: 117 nsWR: R, Iteration:88 data:00110001 00110010 00110011 00110100 00110101
At time: 130 nsWR: R, Iteration:99 data:00110001 00110010 00110011 00110100 00110101
At time: 143 nsWR: W, Iteration:110 data:01110011 01110100 01110101 01110110 01110111
```

Transaction Level Modeling

- Introduction
- + Processing Elements
- + Data
- – Abstract Communications
  - + Coding Styles
    - Loosely Timed
    - Approximately Timed
  - + TLM-2.0 Transport Interfaces
    - Blocking Transport
      - Path (Forward & Return)
      - Socket (Simple Socket)
    - Simple Transport Example
    - Non-blocking Transport
      - Path (Forward, Backward & Return)
      - Socket (Tagged & Multi Socket)
      - Phases & Base Protocol
  - + Complete System

Transaction Level Modeling

- Introduction
- + Processing Elements
- + Data
- – Abstract Communications
  - + Coding Styles
  - + TLM-2.0 Transport Interfaces
    - Blocking Transport
    - Simple Transport Example
    - Non-blocking Transport
- + Complete System



Simple Transport Example

- Example 3: Simple Transport: *interfaceClasses*

interfaceClasses.h

simpleTransport_Main.cpp* | interfaceClasses.cpp | channelTransport.h | interfaceClasses.h

Simple Transport | (Global Scope)

```
1   #include <systemc.h>
2
3   class transport_if : virtual public sc_interface
4   {
5       public:
6           bool loading;
7           sc_event INCOMPLETE;
8           sc_event COMPLETE;
9           virtual void transport(sc_lv<8> &) = 0;
10  };
11
```

Simple Transport Example

- Example 3: Simple Transport: *channelTransport*

channelTransport.h

simpleTransport_Main.cpp* | simpleTransport.cpp | interfaceClasses.cpp | channelTransport.h

Simple Transport | (Global Scope)

```
1   #include "interfaceClasses.h"
2
3   class channelTransport : public transport_if
4   {
5       sc_lv<8> saved;
6       sc_event put_event, get_event;
7       public:
8           channelTransport()  {};
9           ~channelTransport() {};
10          void transport (sc_lv<8> &data);
11  };
```

Simple Transport Example

- Example 3: Simple Transport: *channelTransport*

channelTransport.cpp

interfaceClasses.h | simpleTransport.h | simpleTransport_TB.h | channelTransport.cpp

Simple Transport | (Global Scope)

```
1   #include "channelTransport.h"
2
3   void channelTransport::transport(sc_lv<8> &data) {
4       if (loading == true) {
5           saved = data;
6           wait(10, SC_NS);
7       }
8       else {
9           data = saved;
10          wait(15, SC_NS);
11      }
12  }
```

Simple Transport Example

- Example 3: Simple Transport: *simpleTransport*

simpleTransport.h

simpleTransport_Main.cpp* | channelTransport.h | interfaceClasses.h | simpleTransport.h

Simple Transport | (Global Scope)

```
1   #include "channelTransport.h"
2
3   class transmitter : public sc_module {
4   public:
5       sc_port<transport_if> out;
6
7       SC_CTOR(transmitter) {
8           SC_THREAD (transportingOut);
9       }
10      void transportingOut();
11  };
12
13  class receiver : public sc_module {
14  public:
15      sc_export<transport_if> in;
16      channelTransport* insideTargetChannel;
17
18      SC_CTOR(receiver) {
19          insideTargetChannel = new channelTransport;
20          in(*insideTargetChannel);
21          SC_THREAD(transportingIn);
22      }
23      void transportingIn();
24  };
25
```

Simple Transport Example

simpleTransport_TB.h | channelTransport.cpp | interfaceClasses.cpp | simpleTransport.cpp

Simple Transport | (Global Scope)

```
1   #include "simpleTransport.h"
2
3   void transmitter::transportingOut() {
4       int i;
5       sc_lv<8> dataToPut;
6       out->loading = false;
7       for (i=0; i<27; i++)
8       {
9           dataToPut = (sc_lv<8>) i;
10          out->loading = true;
11          out->transport(dataToPut);
12          out->INCOMPLETE.notify();
13          cout << "Data: (" << dataToPut << ") was transmitted at: "
14              << sc_time_stamp() << '\n';
15          wait(out->COMPLETE);
16      }
17  }
18
19  void receiver::transportingIn() {
20      sc_lv<8> dataThatGot;
21      int i; for (i=0; i<27; i++)
22      while (1)
23      {
24          wait(in->INCOMPLETE);
25          in->loading = false;
26          in->transport(dataThatGot);
27          in->COMPLETE.notify();
28          cout << "Data: (" << dataThatGot << ") was received at: "
29              << sc_time_stamp() << '\n';
30      }
31  }
32
```

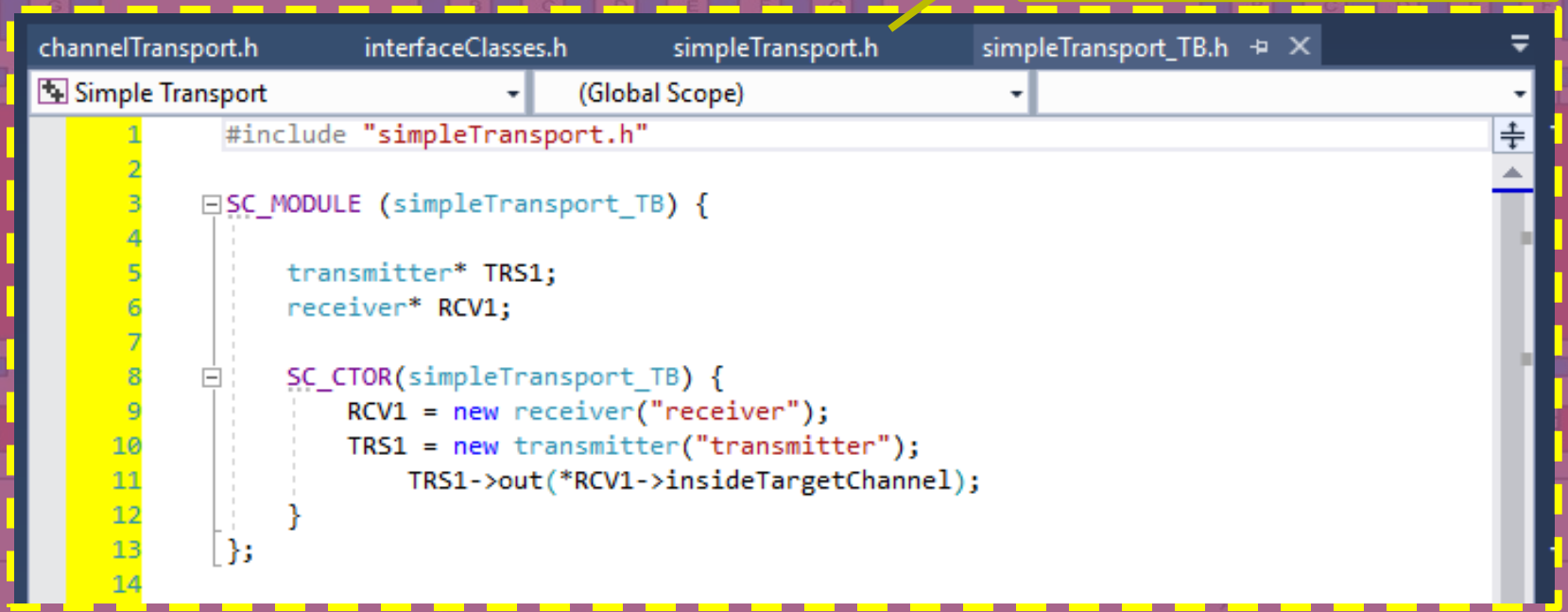
simpleTransport.cpp

- Example 3: Simple Transport: *simpleTransport*

Simple Transport Example

- Example 3: Simple Transport: Testbench

simpleTransport_TB.h



Simple Transport Example

- Example 3: Simple Transport: Main

simpleTransport_Main.cpp

simpleTransport_Main.cpp* | simpleTransport.cpp | interfaceClasses.cpp | channelTransport.h

Simple Transport | (Global Scope)

```
1   #include "simpleTransport_TB.h"
2   int sc_main(int argc, char *argv[]) {
3     simpleTransport_TB SPG1("simpleTransport_TB");
4     sc_start (900, SC_NS);
5     return 0;
6   }
```

Simple Transport Example

- Example 3: Simple Transport: Output

```
        SystemC 2.3.2-Accellera --- Nov 16 2018 05:36:55
        Copyright (c) 1996-2017 by all Contributors,
        ALL RIGHTS RESERVED
Data: (00000000) was transmitted at: 10 ns
Data: (00000000) was received at: 25 ns
Data: (00000001) was transmitted at: 35 ns
Data: (00000001) was received at: 50 ns
Data: (00000010) was transmitted at: 60 ns
Data: (00000010) was received at: 75 ns
Data: (00000011) was transmitted at: 85 ns
Data: (00000011) was received at: 100 ns
Data: (00000100) was transmitted at: 110 ns
Data: (00000100) was received at: 125 ns
Data: (00000101) was transmitted at: 135 ns
Data: (00000101) was received at: 150 ns
Data: (00000110) was transmitted at: 160 ns
Data: (00000110) was received at: 175 ns
Data: (00000111) was transmitted at: 185 ns
Data: (00000111) was received at: 200 ns
Data: (00001000) was transmitted at: 210 ns
Data: (00001000) was received at: 225 ns
Data: (00001001) was transmitted at: 235 ns
Data: (00001001) was received at: 250 ns
Data: (00001010) was transmitted at: 260 ns
Data: (00001010) was received at: 275 ns
Data: (00001011) was transmitted at: 285 ns
Data: (00001011) was received at: 300 ns
Data: (00001100) was transmitted at: 310 ns
Data: (00001100) was received at: 325 ns
Data: (00001101) was transmitted at: 335 ns
```

Transaction Level Modeling

- Introduction
+ Processing Elements
+ Data
– Abstract Communications
  + Coding Styles
    - Loosely Timed
    - Approximately Timed
  + TLM-2.0 Transport Interfaces
    - Blocking Transport
      - Path (Forward & Return)
      - Socket (Simple Socket)
    - Simple Transport Example
    - Non-blocking Transport
      - Path (Forward, Backward & Return)
      - Socket (Tagged & Multi Socket)
      - Phases & Base Protocol
+ Complete System

Non-block

- Non-blocking t
  - Includes timing
  - Typically used
  - Called on forw
  - Is appropriate w
    interactions between i and target during the course of each transaction

TLM_ACCEPTED
- Transaction, phase and timing arguments unmodified (ignored) on return
- Target may respond later (depending on protocol)

TLM_UPDATED
- Transaction, phase and timing arguments updated (used) on return
- Target has advanced the protocol state machine to the next state

TLM_COMPLETED
- Transaction, phase and timing arguments updated (used) on return
- Target has advanced the protocol state machine straight to the final phase

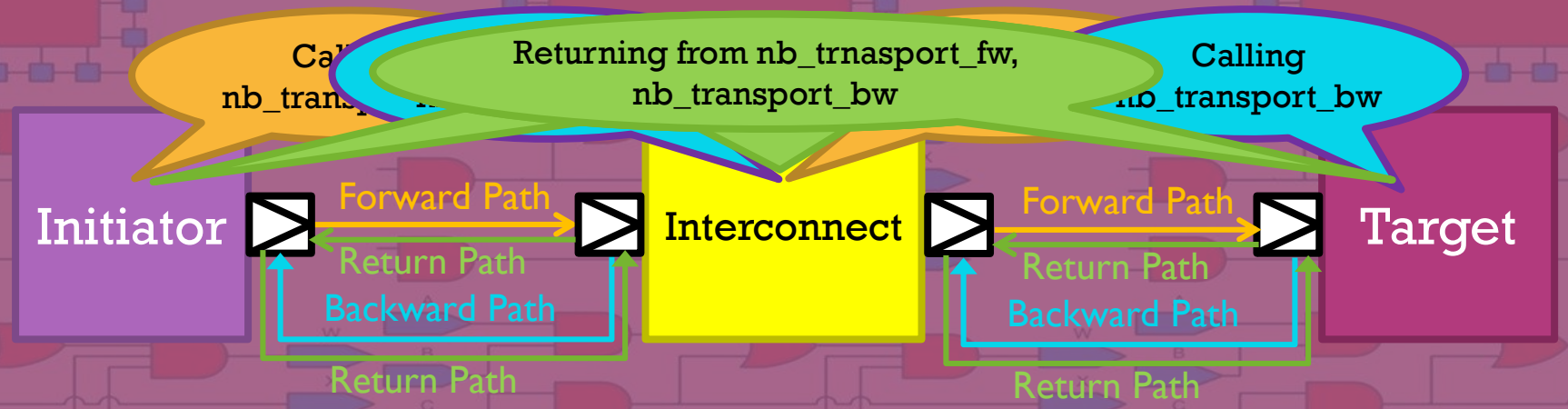
```
enum tlm_sync_enum { TLM_ACCEPTED, TLM_UPDATED, TLM_COMPLETED };

template < typename TRANS= tlm_generic_payload, typename PHASE = tlm_phase>

class tlm_fw_nonblocking_transport_if : public virtual sc_core::sc_interface {
public:
   virtual tlm_sync_enum nb_transport (      TRANS&  trans,
                                             PHASE&  phase,
                                             sc_core::sc_time&  t ) = 0;
};
```

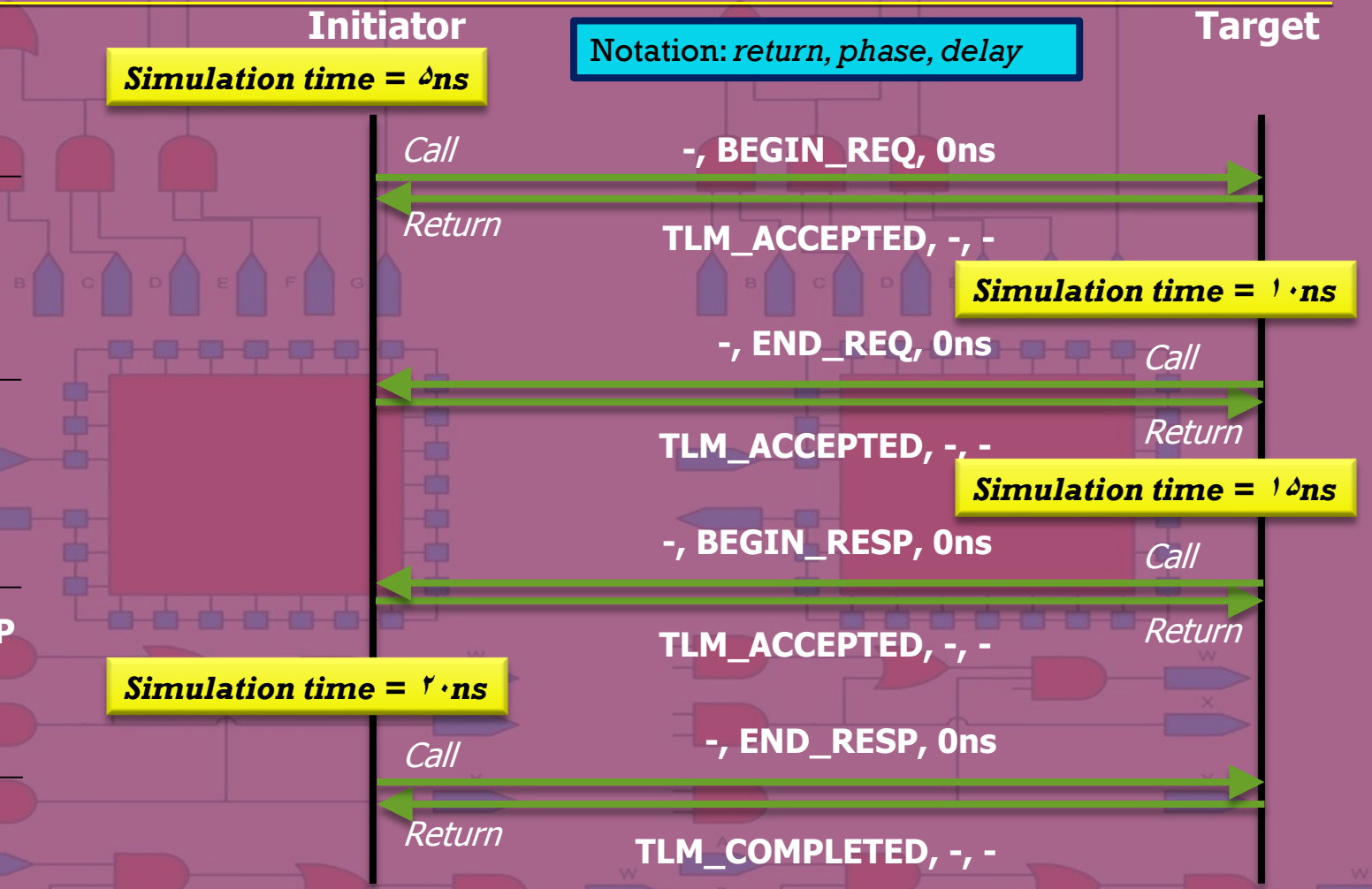
Non-blocking Transport: Path

- Forward path
  - Is the calling path by which an initiator or interconnect component makes interface method calls forward in the direction of another interconnect component or the target
- Backward path
  - Is the calling path by which a target or interconnect component makes interface method calls back in the direction of another interconnect component or the initiator
- Return path
  - Is the return path by which an initiator, target or interconnect component returns immediately from the interface method that has been called



Non-blocking Transport: Path

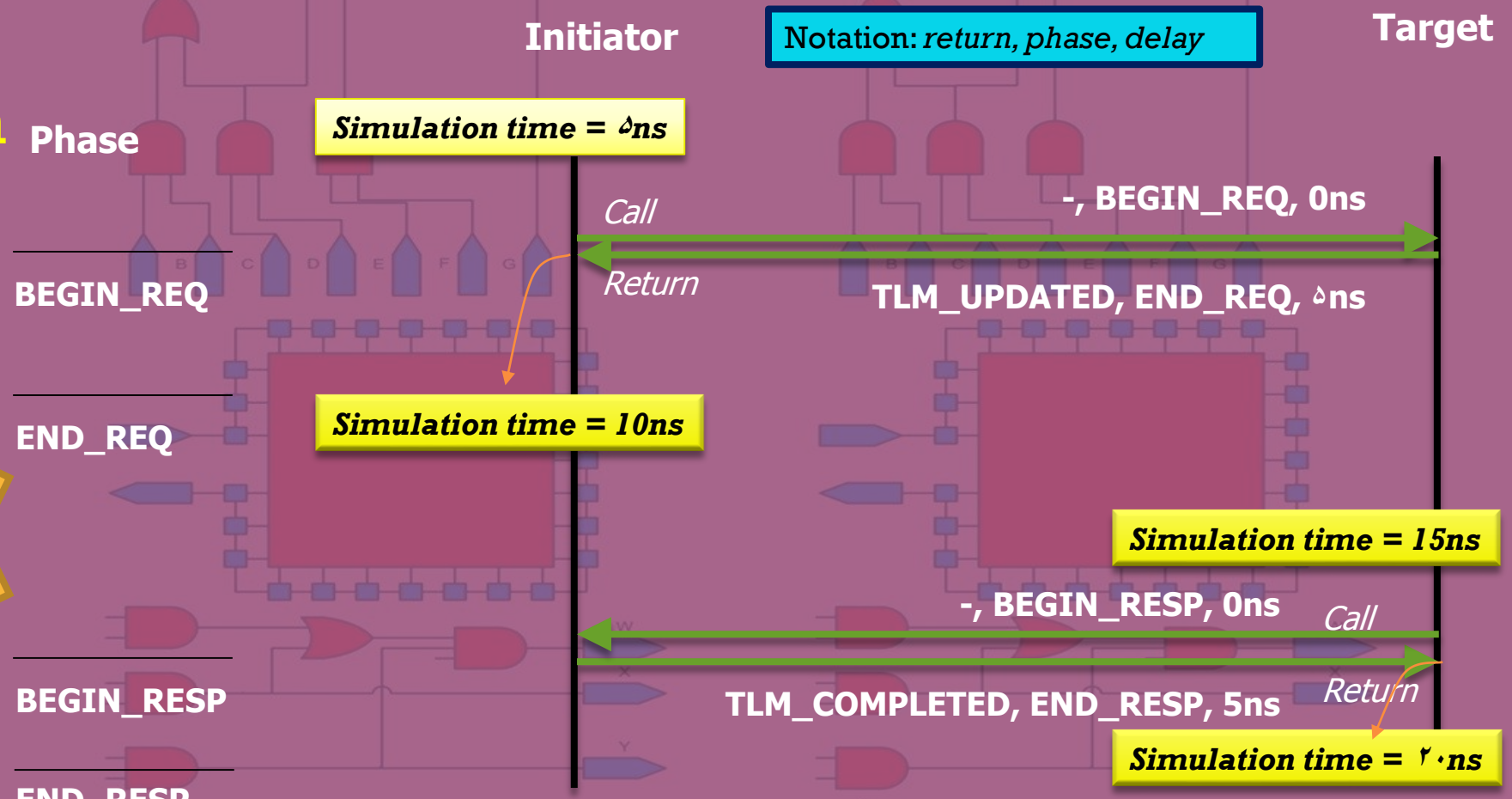
- Timing Diagram: Using The Backward Path



Transaction accepted now, caller asked to wait

Non-blocking Transport: Path

- Timing Diagram: Using The Return Path



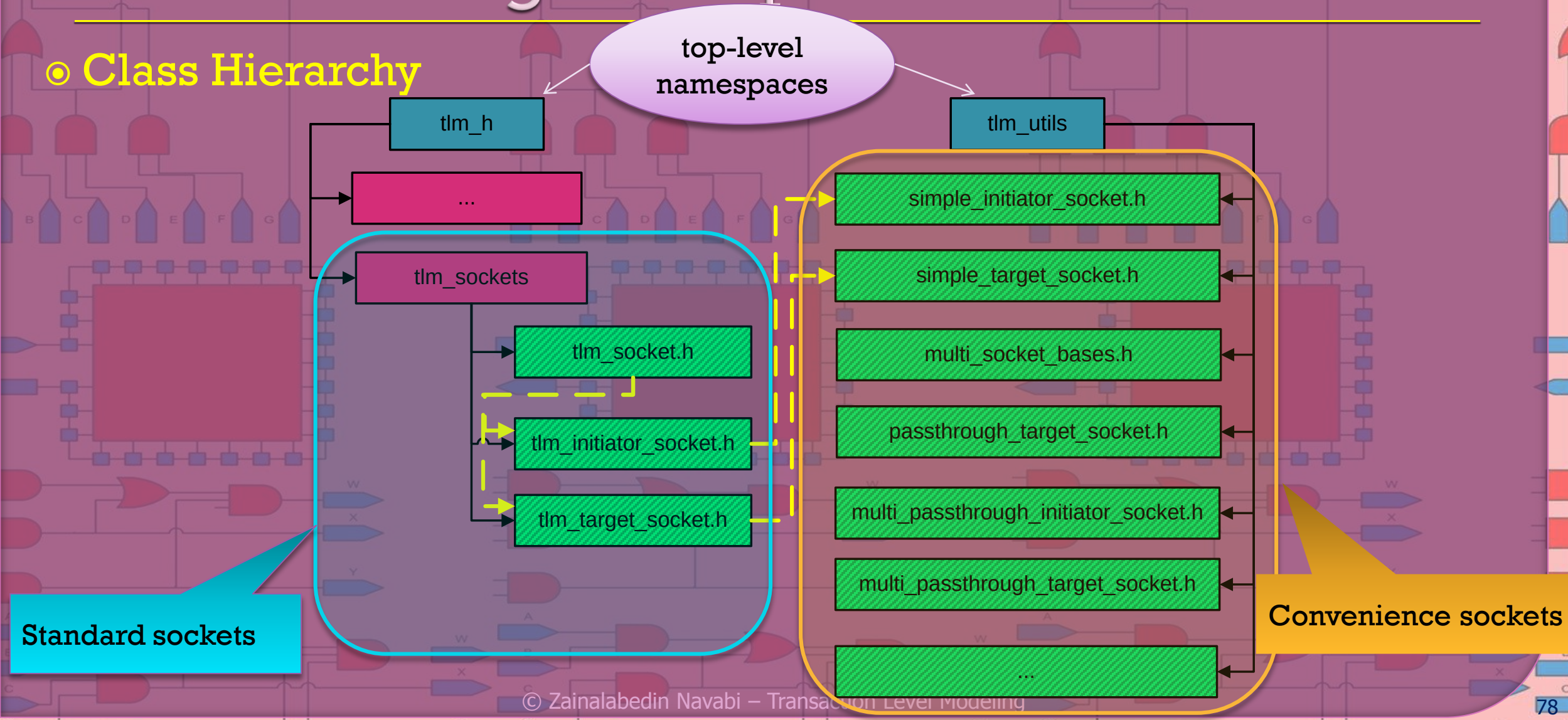
Callee annotates delay to next transition, caller waits

Non-blocking Transport: Sockets

- Combine a port with an export
- Provide methods to bind port and export of both the forward and backward paths in a single call
- Group core interfaces for both the forward and backward paths together into a single object
- Offer strong type checking when binding sockets parameterized with incompatible protocol types
- The classes **tlm_initiator_socket** and **tlm_target_socket** are typically used directly by applications
  - Belong to the interoperability layer of the TLM-2.0 standard

Non-blocking Transport: Sockets

- Class Hierarchy



Non-blocking Transport: Convenience Sockets

- A family of derived socket classes provided in the utilities
  - Are derived from the classes *tlm_initiator_socket* and *tlm_target_socket*
- Additional functionalities
  - Register callbacks: Provides methods to register callbacks for incoming interface method calls
  - Tagged: Incoming interface method calls are tagged with an *id* to indicate the socket through which they arrived
  - Multi-ports: A single initiator socket can be bound to multiple target sockets and vice versa

Non-blocking Transport: Convenience Sockets

- Simple Socket
  - See Slides 52 and 53

Non-blocking Transport: Convenience Sockets

- **Tagged Sockets**

  Incorporates a numerical *tag*
  - To identify the socket in use
  - To allow the callback to identify through which socket the incoming call arrived
    - Allows a single callback function to handle multiple sockets, with the tag identifying the socket which caused the callback to be invoked
    - Useful in the case where the same callback method is registered with multiple initiator sockets or multiple target sockets
  - Is specified when the callback is registered
  - Is an argument of the callback method

Non-blocking Transport: Convenience Sockets

Simple_initiator_socket.h: tagged socket class

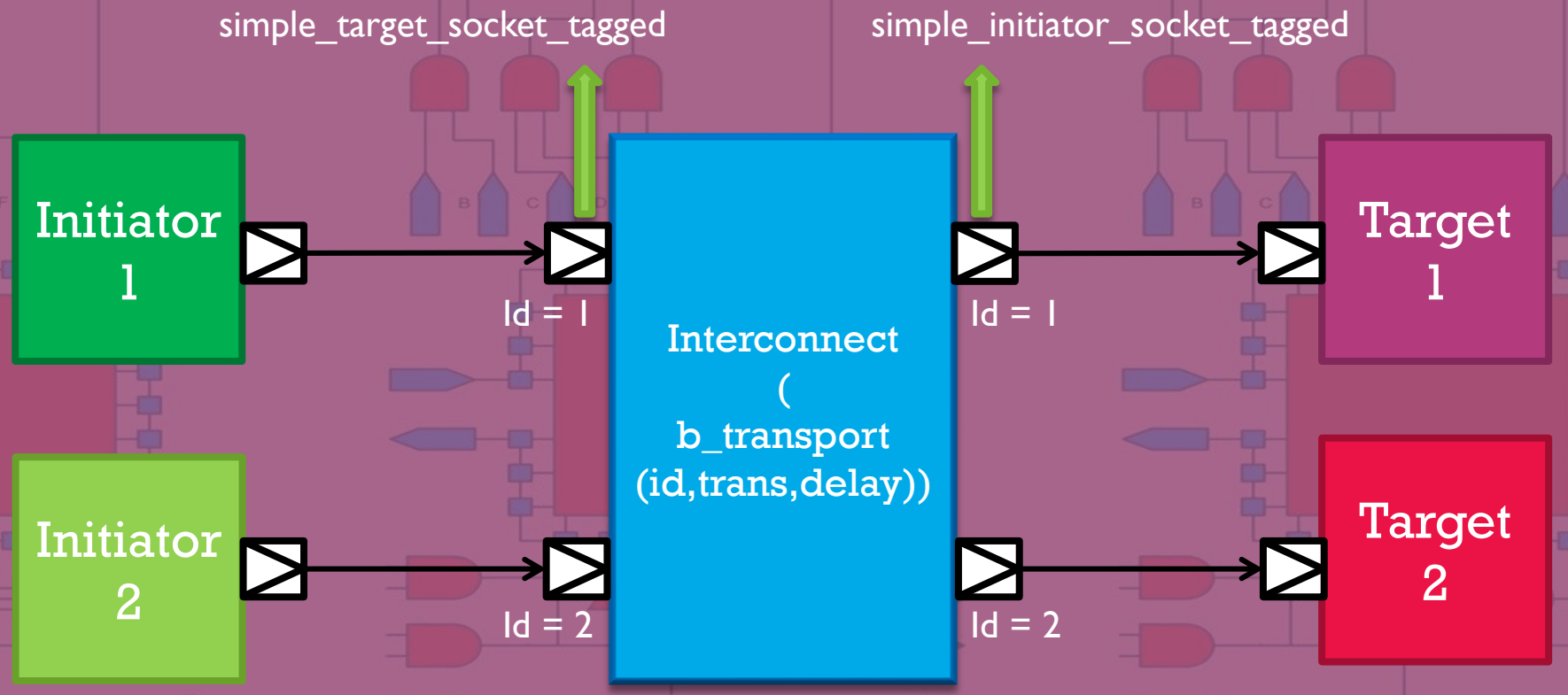
```
151 template <typename MODULE,
152           unsigned int BUSWIDTH = 32,
153           typename TYPES = tlm::tlm_base_protocol_types>
154 class simple_initiator_socket_tagged :
155   public tlm::tlm_initiator_socket<BUSWIDTH, TYPES>
156 {
157 public:
158   typedef typename TYPES::tlm_payload_type             transaction_type;
159   typedef typename TYPES::tlm_phase_type               phase_type;
160   typedef tlm::tlm_sync_enum                           sync_enum_type;
161   typedef tlm::tlm_fw_transport_if<TYPES>              fw_interface_type;
162   typedef tlm::tlm_bw_transport_if<TYPES>              bw_interface_type;
163   typedef tlm::tlm_initiator_socket<BUSWIDTH, TYPES>   base_type;
164
165 public:
166   simple_initiator_socket_tagged() { ... }
173   explicit simple_initiator_socket_tagged(const char* n) { ... }
180   void register_nb_transport_bw(MODULE* mod,
181                                 sync_enum_type (MODULE::*cb)(int,
182                                                              transaction_type&,
183                                                              phase_type&,
184                                                              sc_core::sc_time&),
185                                 int id) { ... }
191   void register_invalidate_direct_mem_ptr(MODULE* mod,
192                                           void (MODULE::*cb)(int, sc_dt::uint64, sc_dt::uint64),
193                                           int id) { ... }
199 private:
200   class process { ... };
284 private:
285   process m_process;
286 };
```

The tag that is used for identifying the socket of the incoming transaction

⊙ Tagged Socket: Class Definition

Non-blocking Transport: Convenience Sockets

- Tagged Socket:



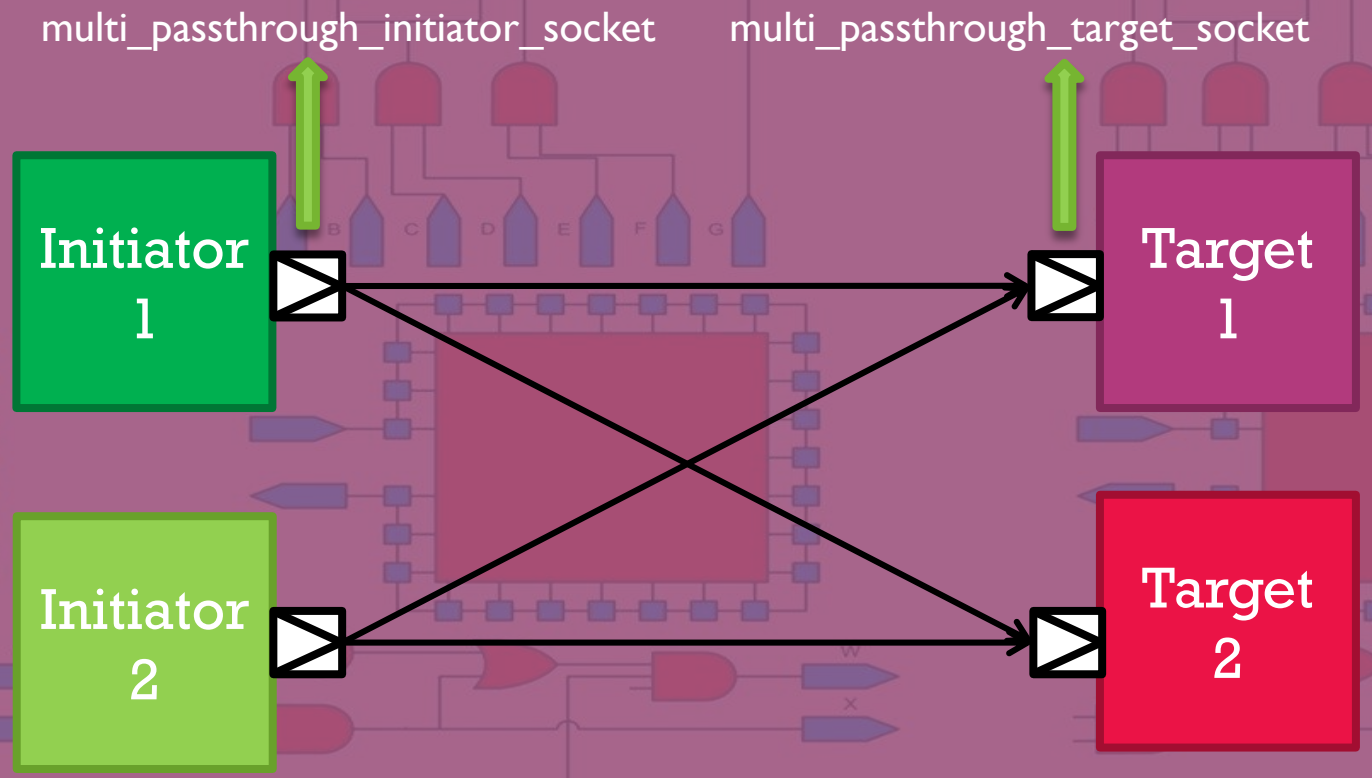
Distinguish origin of incoming transactions using socket *id*

Non-blocking Transport: Convenience Sockets

- **Multi Socket**
  - A variation on the tagged simple sockets
  - Permit a single socket to be bound to multiple sockets on other components
  - Use multi-port index number as the tag
    - Is able to identify from which socket on another component an incoming interface method call arrives

Non-blocking Transport: Convenience Sockets

- Multi Socket:



Many-to-Many socket binding
Method calls tagged with multi-port index value

Non-blocking Transport: Phase & Base Protocol

- Phase
  - Class **tlm_phase** is the default phase type used by the non-blocking transport interface class templates and the base protocol
  - Base protocol phases:
    - BEGIN_REQ, END_REQ, BEGIN_RESP, and END_RESP
  - Extending the set of four phases provided by **tlm_phase_enum**:
    - Using the macro DECLARE_EXTENDED_PHASE
  - For maximal interoperability, an application should only use the four phases of **tlm_phase_enum**

Non-blocking Transport: Phase & Base Protocol

⊙ Phases: Class Definition

tlm_phase.h

```
25  namespace tlm {
26
27  //enum tlm_phase { BEGIN_REQ, END_REQ, BEGIN_RESP, END_RESP };
28
29  enum tlm_phase_enum { UNINITIALIZED_PHASE=0, BEGIN_REQ=1, END_REQ, BEGIN_RESP, END_RESP };
30
    inline unsigned int create_phase_number() { ... }
    inline std::vector<const char*>& get_phase_name_vec() { ... }
41  class tlm_phase{
42  public:
43    tlm_phase(): m_id(0) {}
44    tlm_phase(unsigned int id): m_id(id){}
45    tlm_phase(const tlm_phase_enum& standard): m_id((unsigned int) standard){}
46    tlm_phase& operator=(const tlm_phase_enum& standard){m_id=(unsigned int)standard; return *this;}
47    operator unsigned int() const{return m_id;}
48
49  private:
50    unsigned int m_id;
51  };
52  ...
67  #define DECLARE_EXTENDED_PHASE(name_arg) \
68  ...
78
79  } // namespace tlm
```

The enumeration having values corresponding to the four phases of the base protocol: BEGIN_REQ, END_REQ, BEGIN_RESP, and END_RESP

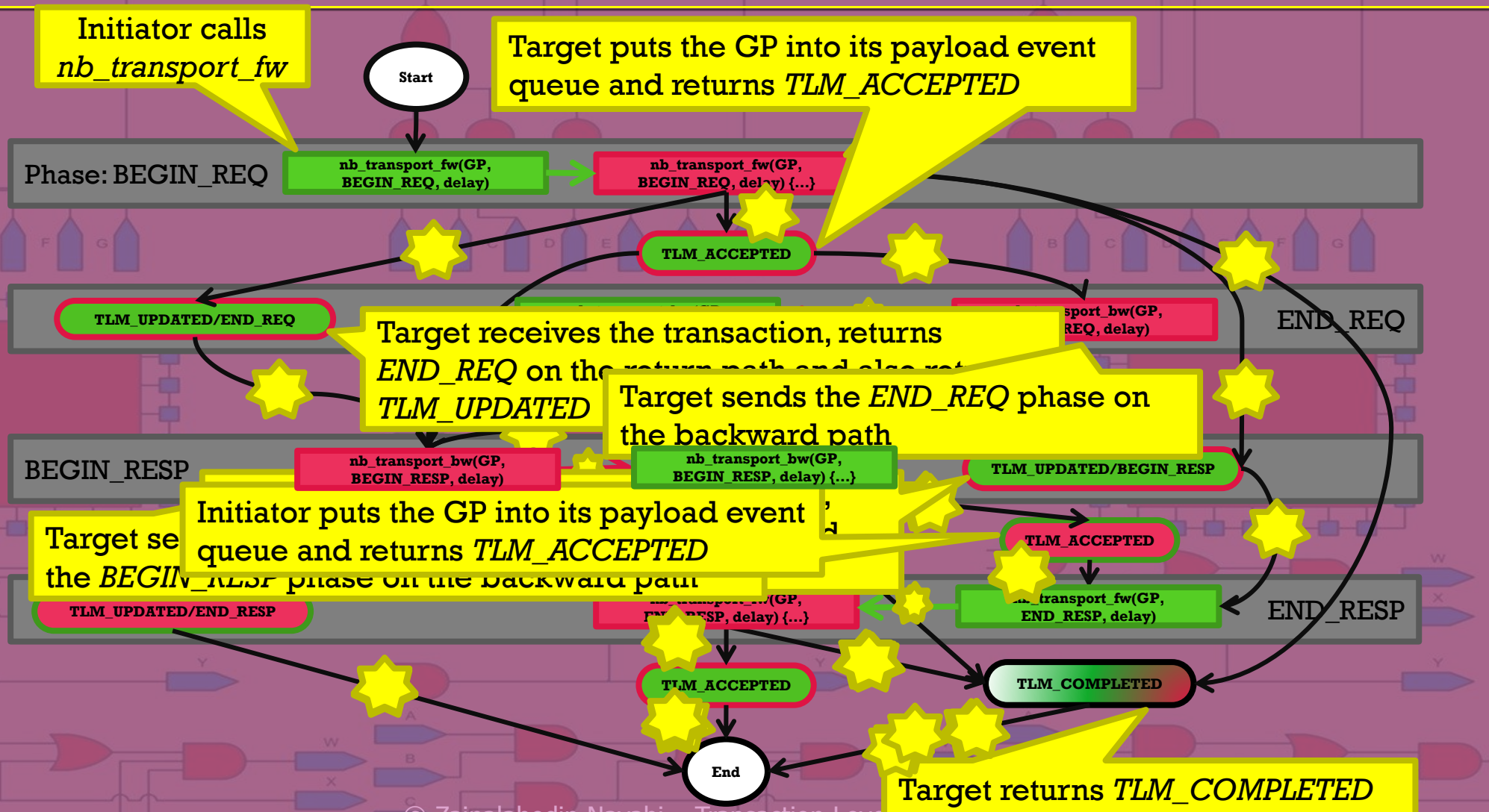
Is used to extend the four declared phases

Non-blocking Transport: Phase & Base Protocol

- Base Protocol
  - The base protocol = tlm_generic_payload + tlm_phase
  - consists of a set of rules to ensure maximal interoperability between transaction level models of components

```
namespace tlm {
struct tlm_base_protocol_types
}
typedef tlm_generic_payload tlm_payload_type;
typede tlm_phase tlm_phase_type;
;{
} // namespace tlm
```

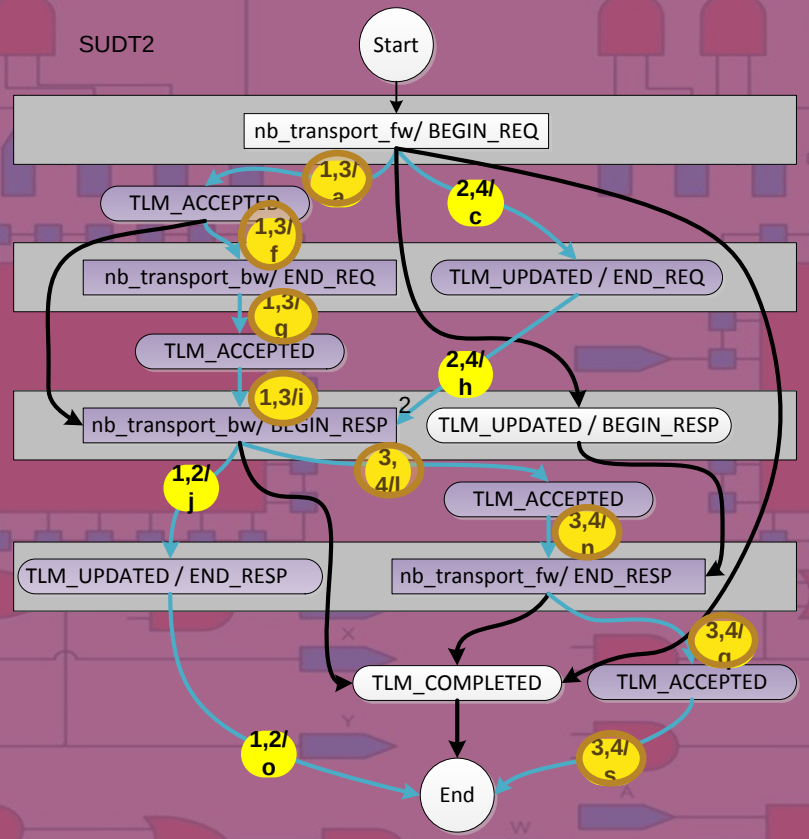
Non-blocking Transport: Phase & Base Protocol



© Zainalabedin Navabi – Transaction Level Modeling

Non-blocking Transport: Phase & Base Protocol

◉ Abstract Communication Scenarios

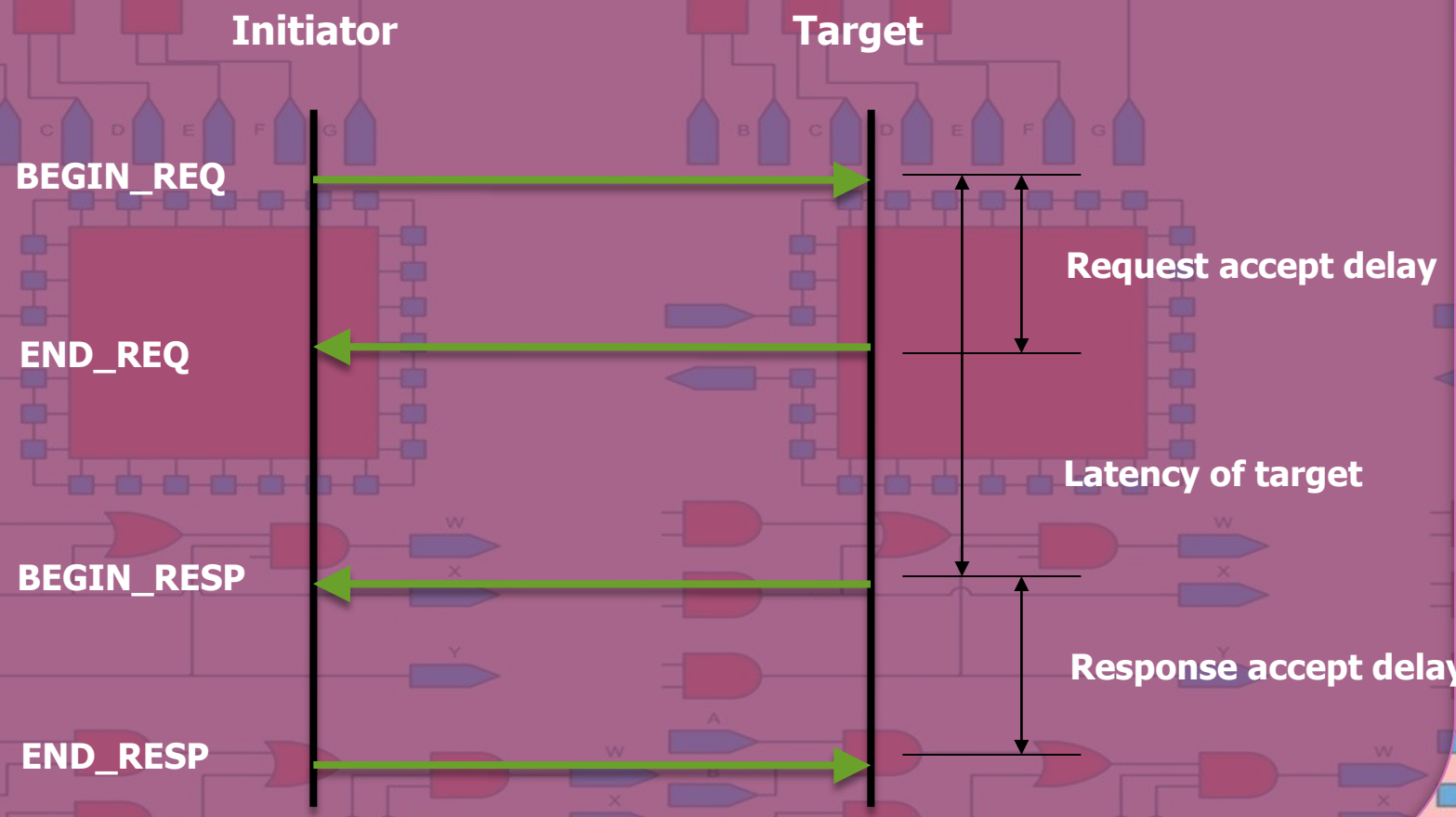


TLM Comm. Scenario	TLM-2.0 base protocol flow: Edge id.							
SUDT2	c	h	j	o				
	a	f	g	i	j	o		
	c	h	l	n	q	s		
	a	f	g	i	l	n	q	s
SUDT3	a	e	j	o				
	a	e	l	n	q	s		
SUDT4	a	e	l	n	q	s		
	a	e	l	n	p	r		
	b	m	q	s				
	b	m	p	r				
SUDT5	d	r						
SUDT7	c	h	k	r				
	a	f	g	i	k	r		
SUDT8	a	e	k	r				
	a	e	l	n	q	s		
SUDT9	a	e	k	r				
	b	m	q	s				
	b	m	p	r				
SUDT10	c	h	l	n	p	r		
	a	f	g	i	l	n	p	r

Non-blocking Transport: Phase & Base Protocol

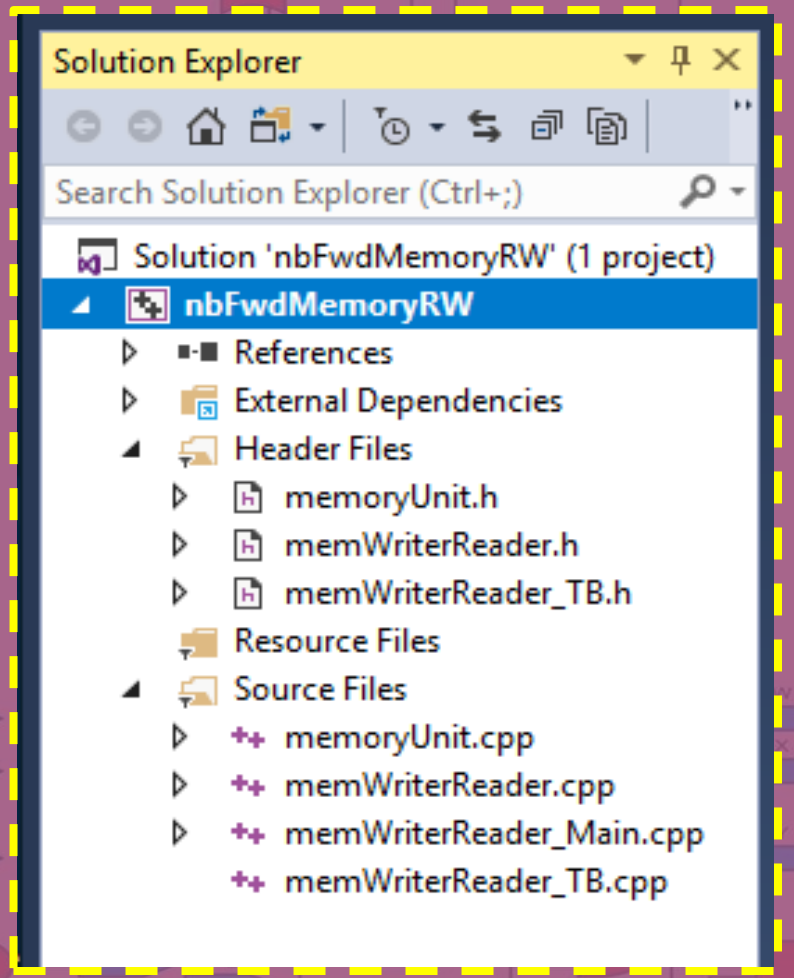
⊙ Base Protocol Timing Parameters and Flow Control

BEGIN_REQ must wait for previous END_REQ, BEGIN_RESP for END_RESP



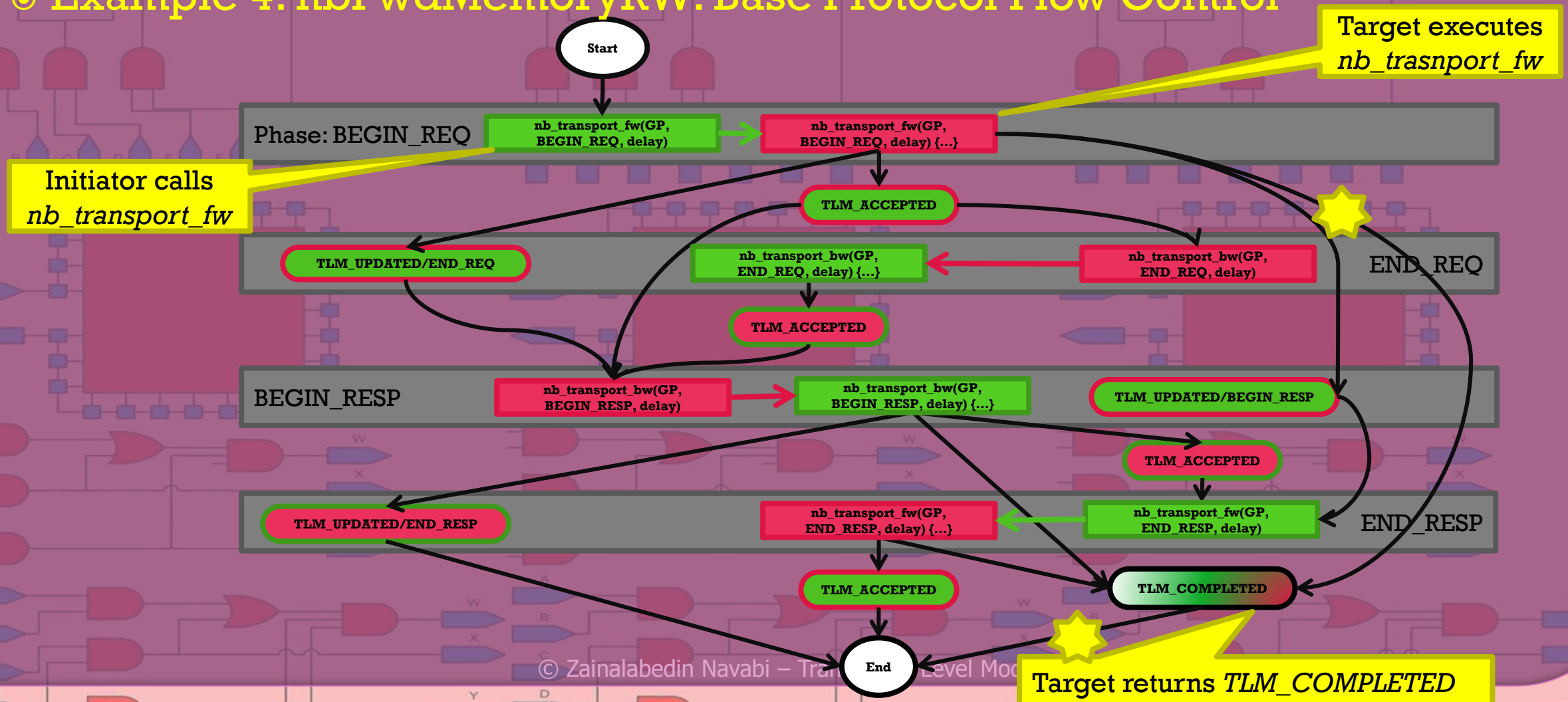
Transaction Level Modeling

- Introduction
- + Processing Elements
- + Data
- – Abstract Communications
  - + Coding Styles
  - + TLM-2.0 Transport Interfaces
    - Blocking Transport
    - Simple Transport Example
    - Non-blocking Transport
      - Path (Forward, Backward & Return)
      - Socket (Tagged & Multi Socket)
      - Phases & Base Protocol
      - nbFwdMemoryRW Example
- + Complete System



Non-blocking Transport: Forward

- Example 4: nbFwdMemoryRW: Base Protocol Flow Control



© Zainalabedin Navabi – Tra Level Mod

Non-blocking Transport: Forward

- Example 4: nbFwdMemoryRW: Initiator, *memWriterReader*

memWriterReader.cpp | memWriterReader_TB.h | memWriterReader.h | memoryUnit.h

nbFwdMemoryRW | (Global Scope)

```
1   #include <systemc.h>
2
3   #include "tlm.h"
4   #include "tlm_utils/simple_initiator_socket.h"
5   #include "tlm_utils/simple_target_socket.h"
6
7   class memWriterReader : public sc_module {
8   public:
9       tlm_utils::simple_initiator_socket<memWriterReader, 32> memWRSocket;
10
11      SC_CTOR(memWriterReader) : memWRSocket("mem_WR_socket"), nBlockWriteRead(0)
12      {
13          nBlockWriteRead = new tlm::tlm_generic_payload();
14          for (int i = 0; i < 4; i++) *(data+i) = i+192;
15          SC_THREAD(nbMemWR);
16      }
17
18      tlm::tlm_generic_payload* nBlockWriteRead;
19      void nbMemWR();
20      void doSomethingGood(tlm::tlm_generic_payload&);
21
22      sc_lv<8> data[5];
23  };
```

memWriterReader.h

Defining the Initiator socket and specializing it for memWriterReader

Initializing memory

Non-blocking Transport: Forward

- Example 4: nbFwdMemoryRW: Initiator, *memWriterReader*

memWriterReader_TB.h | memWriterReader.h | memoryUnit.h | memWriterReader.cpp

nbFwdMemoryRW | (Global Scope)

memWriterReader.cpp

```
#include "memWriterReader.h"

void memWriterReader::nbMemWR()
{
    tlm::tlm_phase forwardPhase;
    sc_time processTime; // Processing time of initiator prior to call

    processTime = sc_time(0, SC_PS);
    for (int i = 0; i < 111; i = i + 11){
        tlm::tlm_command cmd = (tlm::tlm_command)(rand() % 2);
        if (cmd == tlm::TLM_WRITE_COMMAND) {
            data[0] = (sc_lv<8>) (i+5);
            data[1] = (sc_lv<8>) (i+6);
            data[2] = (sc_lv<8>) (i+7);
            data[3] = (sc_lv<8>) (i+8);
            data[4] = (sc_lv<8>) (i+9);
        }

        nBlockWriteRead->set_command( cmd );
        nBlockWriteRead->set_address( i );
        nBlockWriteRead->set_data_ptr( (unsigned char*) data );
        nBlockWriteRead->set_data_length( 5 );
        nBlockWriteRead->set_streaming_width( 5 );
        nBlockWriteRead->set_byte_enable_ptr( 0 );
        nBlockWriteRead->set_dmi_allowed( false );
        nBlockWriteRead->set_response_status( tlm::TLM_INCOMPLETE_RESPONSE );

        forwardPhase = tlm::BEGIN_REQ;

        cout << (cmd ? 'W' : 'R') << ", @" << i << " data:";
        sc_lv<8> vv;
        for(int j=0; j<5; j++) {vv=data[j]; cout << vv << " ";}
        cout << " @time " << sc_time_stamp() <<" delay=" << processTime << '\n';

        tlm::tlm_sync_enum returnStatus;
        returnStatus = memWRSocket->
            nb_transport_fw(*nBlockWriteRead, forwardPhase, processTime);

        if (returnStatus == tlm::TLM_COMPLETED)
            doSomethingGood( *nBlockWriteRead, processTime );
    }
}
```

Set GP parameters

Setting the forward phase for starting a new transaction

Defining the *returnStatus* variable

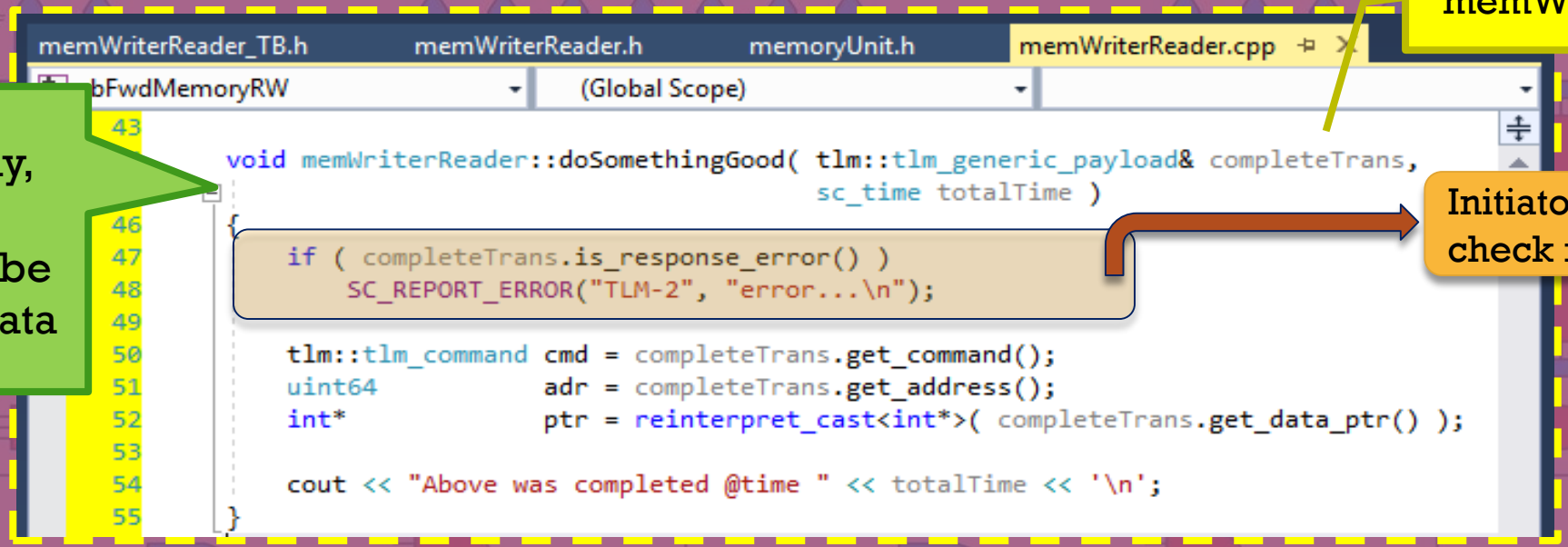
Calling the *nb_transport_fw* method

The initiator can process the data of the incoming transaction in the case of receiving TLM_COMPLETED

Non-blocking Transport: Forward

- Example 4: nbFwdMemoryRW: Initiator, *memWriterReader, Cont.*

memWriterReader.cpp



The data is ready, an appropriate processing can be started on the data

Initiator obliged to check response status

Non-blocking Transport: Forward

- Example 4: nbFwdMemoryRW: Target, *memoryUnit*

memWriterReader.cpp | memWriterReader_TB.h | memWriterReader.h | memoryUnit.h

nbFwdMemoryRW | (Global Scope)

```
1   #include <systemc.h>
2
3   #include "tlm.h"
4   #include "tlm_utils/simple_initiator_socket.h"
5   #include "tlm_utils/simple_target_socket.h"
6
7   class memoryUnit : public sc_module {
8   public:
9       tlm_utils::simple_target_socket<memoryUnit, 32> memSocket;
10
11      static const int SIZE=256;
12
13      SC_CTOR(memoryUnit) : memSocket("memory_side_socket") {
14          memSocket.register_nb_transport_fw(this, &memoryUnit::nb_transport_fw);
15
16          // Initialize memory
17          for (int i = 0; i < SIZE; i++)
18              memArray[i] = (sc_lv<8>) (i%256 + 192);
19      }
20
21      virtual tlm::tlm_sync_enum nb_transport_fw( tlm::tlm_generic_payload&,
22                                                  tlm::tlm_phase&, sc_time& );
23      sc_lv<8> memArray[SIZE];
24  };
```

memoryUnit.h

Defining the target socket and specializing it for memoryUnit

Register callback for incoming nb_transport_fw interface method call

Initializing memory

Non-blocking Transport: Forward

memoryUnit.cpp

```
memWriterReader.h   memoryUnit.h   memWriterReader.cpp
nbFwdMemoryRW                      (Global Scope)

#include "memoryUnit.h"

tlm::tlm_sync_enum memoryUnit::nb_transport_fw
      (tlm::tlm_generic_payload& receivedTrans,
       tlm::tlm_phase& phase, sc_time& delay ){

   tlm::tlm_command cmd = receivedTrans.get_command();
   uint64           adr = receivedTrans.get_address();
   unsigned char*   ptr = receivedTrans.get_data_ptr();
   unsigned int     len = receivedTrans.get_data_length();
   unsigned char*   byt = receivedTrans.get_byte_enable_ptr();
   unsigned int     wid = receivedTrans.get_streaming_width();

   if (byt != 0) {
      receivedTrans.set_response_status( tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE );
      return tlm::TLM_COMPLETED;
   }
   if (len > 5 || wid < len) {
      receivedTrans.set_response_status( tlm::TLM_BURST_ERROR_RESPONSE );
      return tlm::TLM_COMPLETED;
   }
   unsigned int i;
   if ( cmd == tlm::TLM_READ_COMMAND )
      for(i=0; i<len; i=i+1) {
         *(ptr+i) = *((unsigned char*) (memArray+adr+i));
      }
   else if ( cmd == tlm::TLM_WRITE_COMMAND )
      for(i=0; i<len; i=i+1) {
         *((unsigned char*) (memArray+adr+i)) = *(ptr+i);
      }

   receivedTrans.set_response_status( tlm::TLM_OK_RESPONSE );
   delay = delay + sc_time(123, SC_NS);
   return tlm::TLM_COMPLETED;
}
```

- Example 4: nbFwdMemoryRW: Target, *memoryUnit*

Obliged to implement read and write commands

Obliged to set response status to indicate successful completion

Should return TLM_COMPLETED according to the protocol

Non-blocking Transport: Forward

- Example 4: nbFwdMemoryRW: Testbench

memWriterReader_TB.h

memWriterReader.cpp | memWriterReader_TB.h | memWriterReader.h | memoryUnit.h

nbFwdMemoryRW | (Global Scope)

```
1   #include "memWriterReader.h"
2   #include "memoryUnit.h"
3
4   SC_MODULE(memWriterReader_TB)
5   {
6       memWriterReader *WR1;
7       memoryUnit    *MU1;
8
9       SC_CTOR(memWriterReader_TB
10      {
11          WR1 = new memWriterReader("WR");
12          MU1 = new memoryUnit("memory");
13          WR1->memWRSocket.bind(MU1->memSocket);
14      }
15  };
```

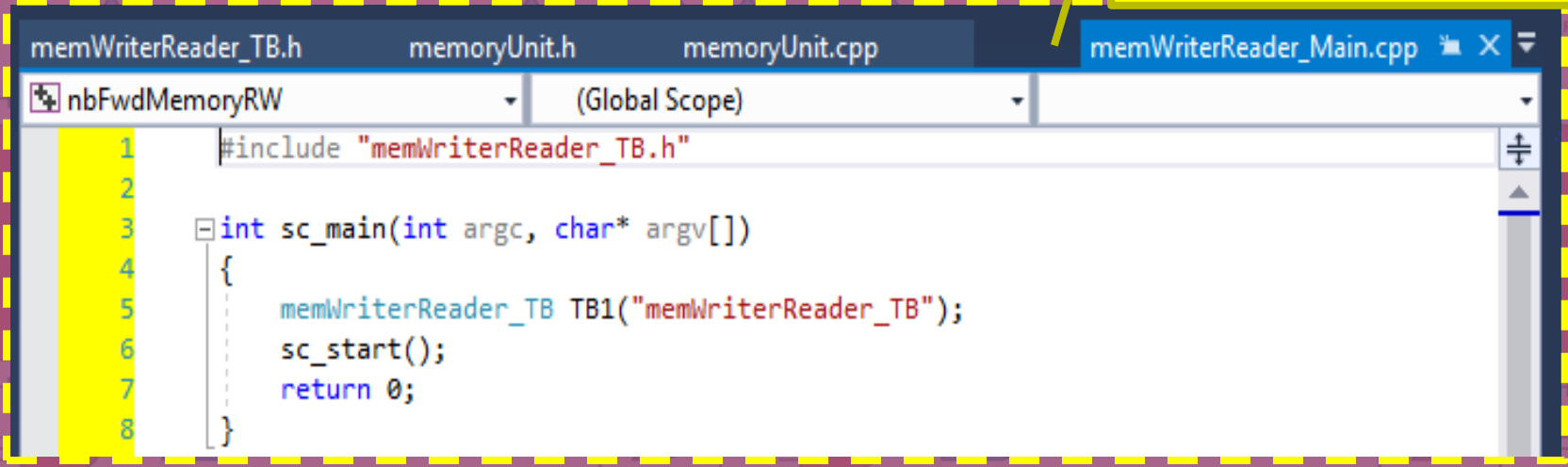
Instantiate components

- One initiator is bound directly to one target with no intervening bus
- Bind initiator socket to target socket

Non-blocking Transport: Forward

- Example 4: nbFwdMemoryRW: Main

memWriterReader_Main.cpp



Non-blocking Transport: Forward

- Example 4: nbFwdMemoryRW: Output

```
        SystemC 2.3.2-Accellera --- Nov 18 2018 08:59:55
        Copyright (c) 1996-2017 by all Contributors,
        ALL RIGHTS RESERVED
W, @0 data:00000101 00000110 00000111 00001000 00001001  @time 0 s delay=0 s
Above was completed @time 123 ns
W, @11 data:00010000 00010001 00010010 00010011 00010100  @time 0 s delay=123 ns
Above was completed @time 246 ns
R, @22 data:00010000 00010001 00010010 00010011 00010100  @time 0 s delay=246 ns
Above was completed @time 369 ns
R, @33 data:00010000 00010001 00010010 00010011 00010100  @time 0 s delay=369 ns
Above was completed @time 492 ns
W, @44 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=492 ns
Above was completed @time 615 ns
R, @55 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=615 ns
Above was completed @time 738 ns
R, @66 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=738 ns
Above was completed @time 861 ns
R, @77 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=861 ns
Above was completed @time 984 ns
R, @88 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=984 ns
Above was completed @time 1107 ns
R, @99 data:00110001 00110010 00110011 00110100 00110101  @time 0 s delay=1107 ns
Above was completed @time 1230 ns
W, @110 data:01110011 01110100 01110101 01110110 01110111  @time 0 s delay=1230 ns
Above was completed @time 1353 ns
```

Acknowledgment

Slides developed by:

Nooshin Nosrati, Ph.D. Student, ECE, University of Tehran