Chapter 7

Transaction Level Modeling (TLM)

Zainalabedin Navabi

Slides updated by Nooshin Nosrati

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?

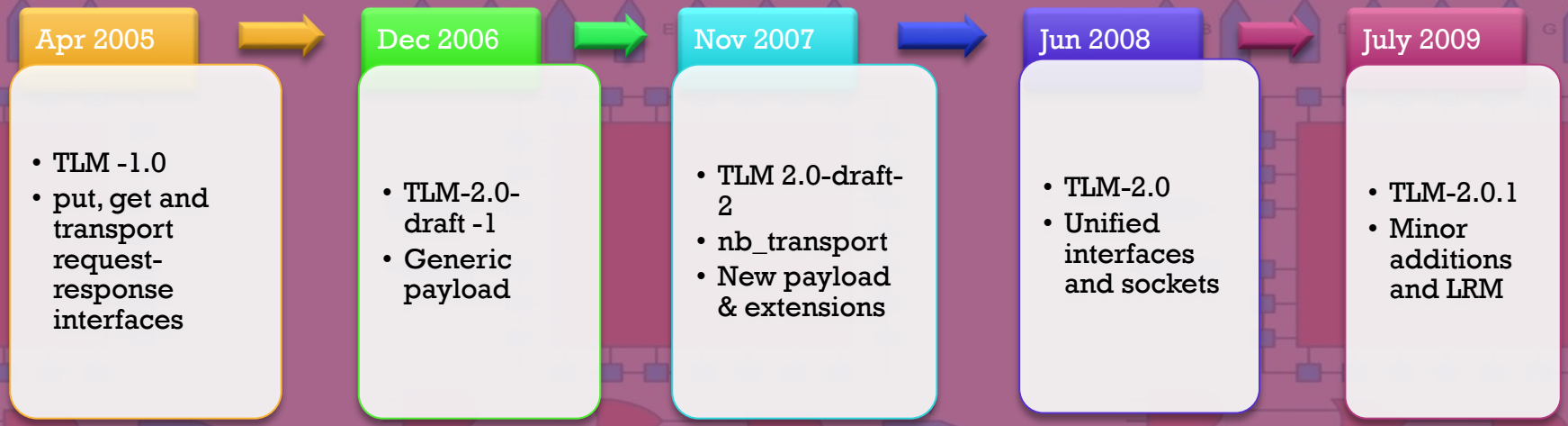
- Higher simulation speed than RTL
- Design space exploration
- Early verification
- HW/SW communication

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

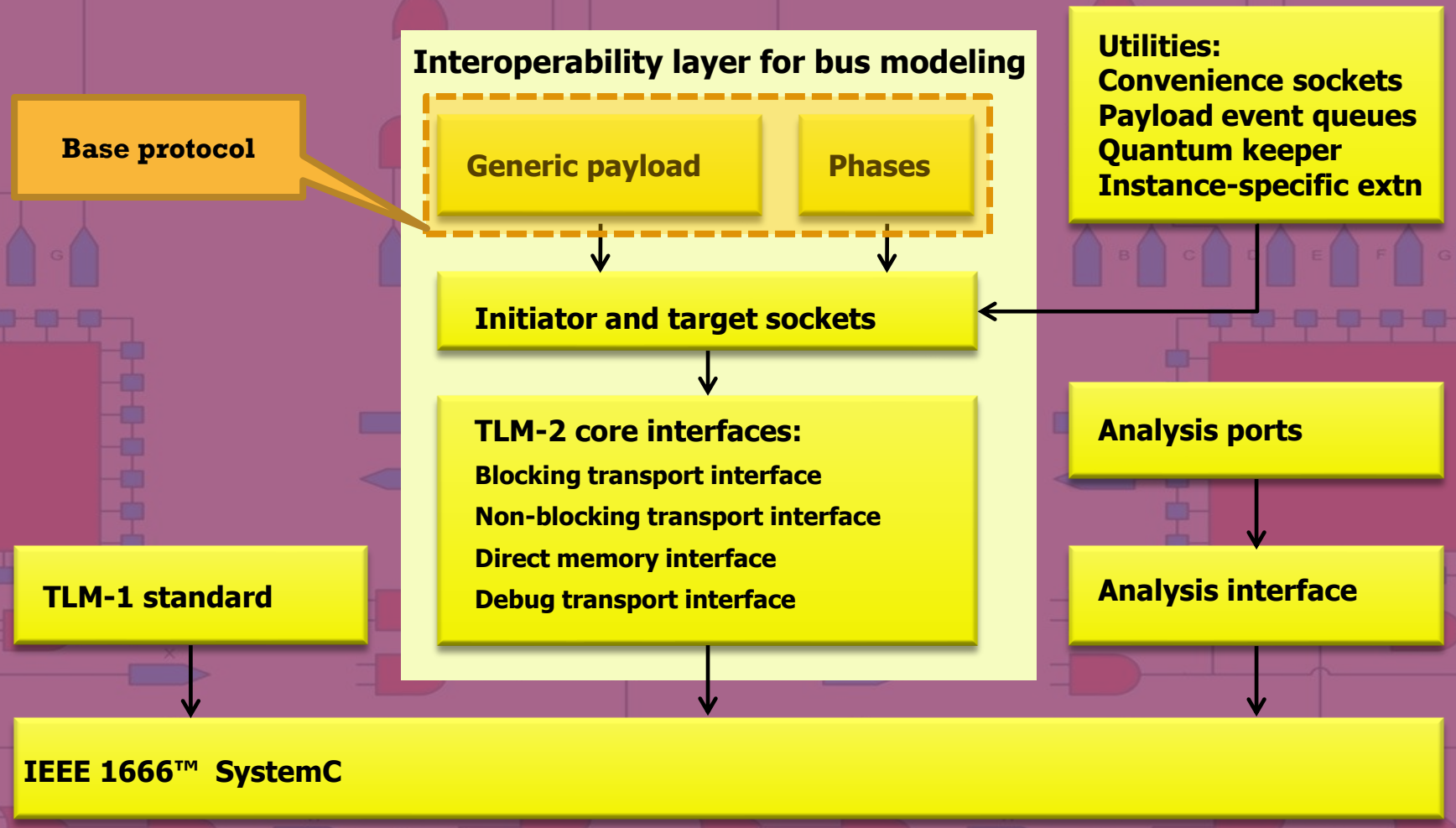
Such as RTL components: Bus, Register, …

OSCI TLM Development



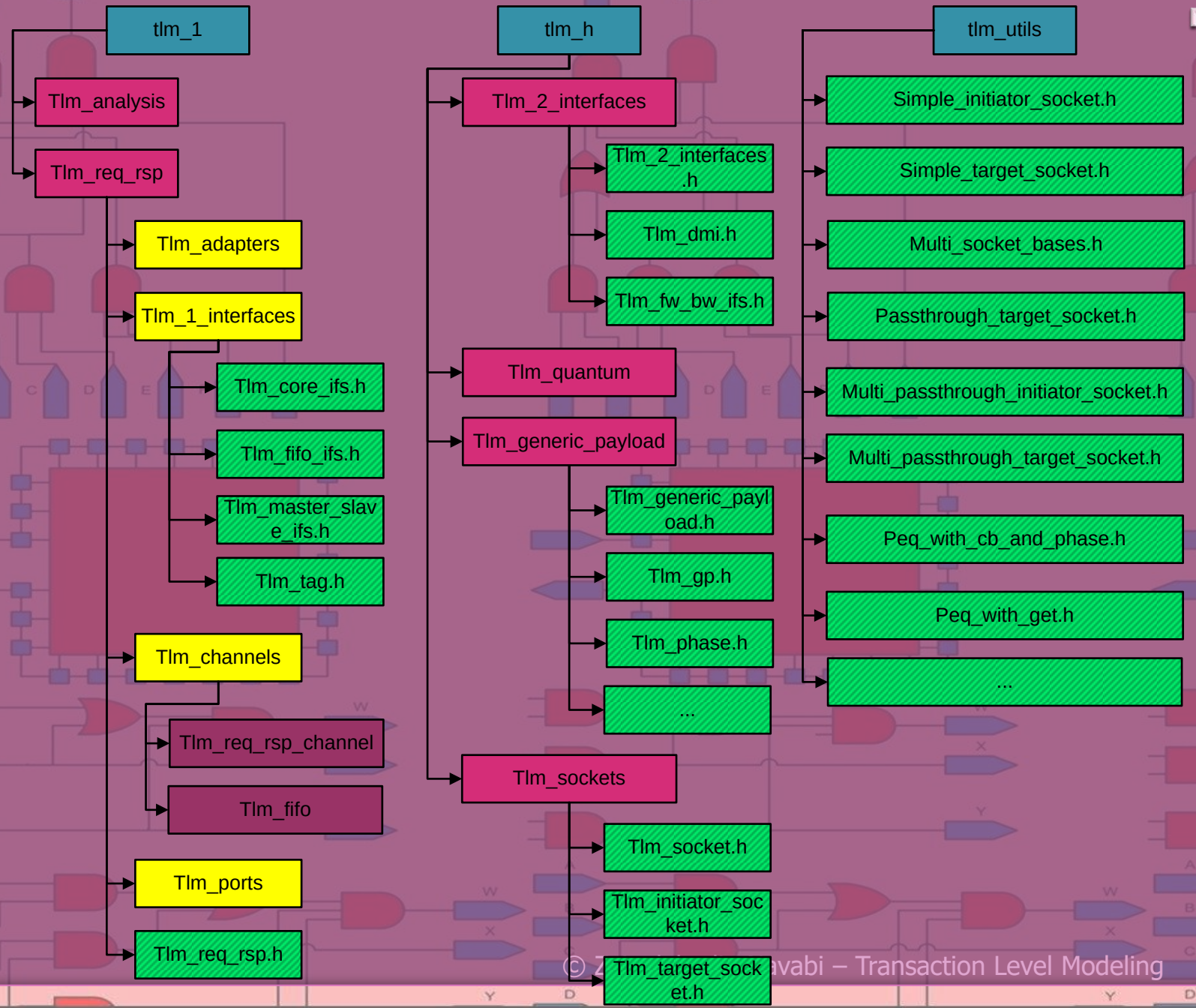
* Open SystemC Initiative, 31 May 2009

TLM-2.0 library Structure



* Open SystemC Initiative, 31 May 2009

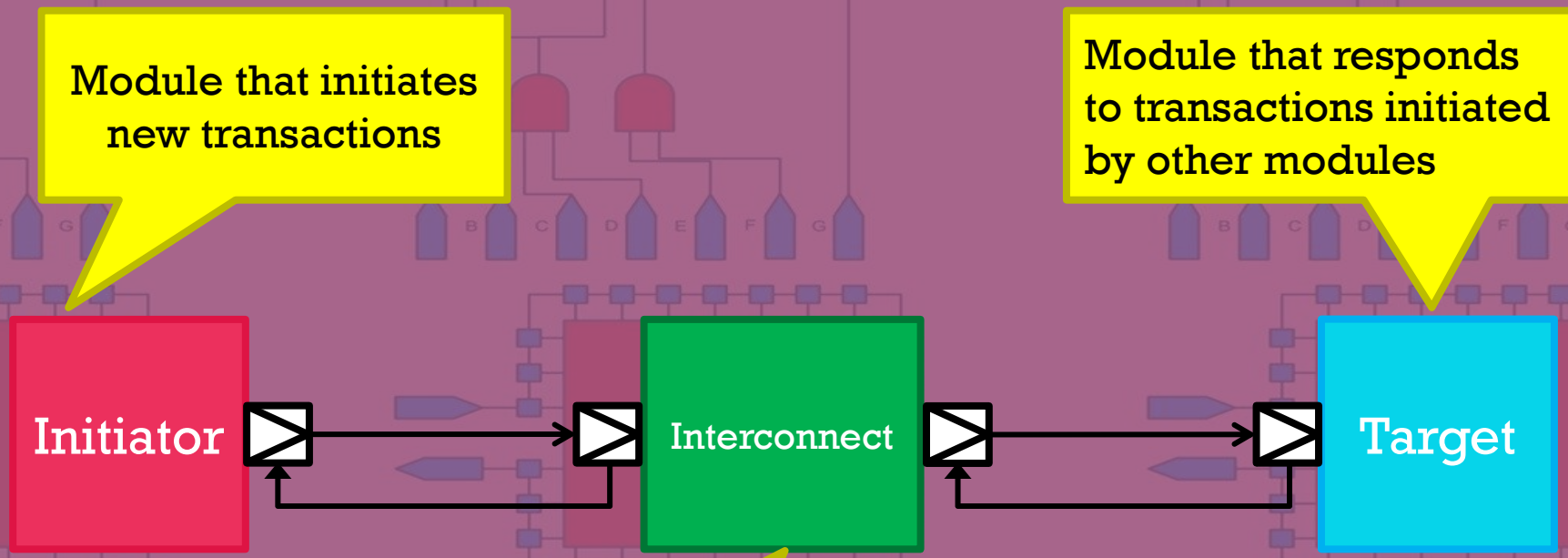
TLM-2.0 Library Hierarchy



Transaction Level Modeling

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

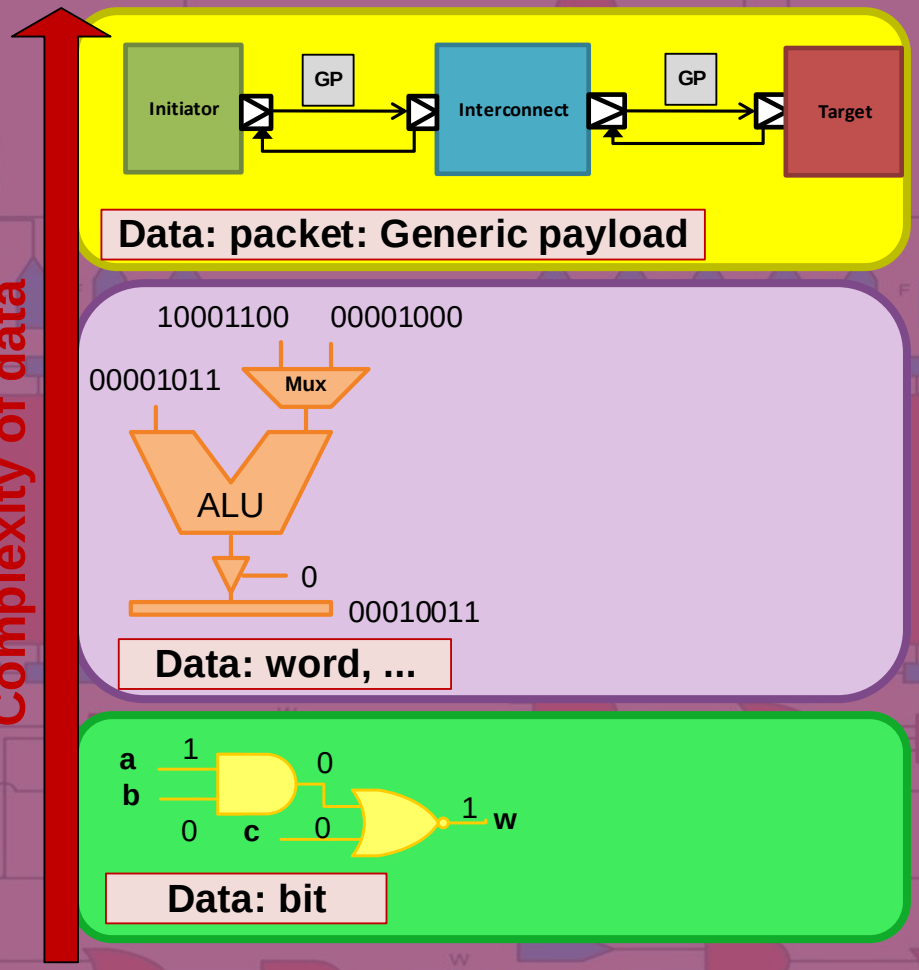
Processing Elements



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
```

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
- may be used to create burst transfers where the address increment between each beat is greater than the number of significant bytes transferred on each beat, or to place words in selected byte lanes of a bus
- The number of elements in the byte enable array shall be given by the byte enable length attribute
- 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

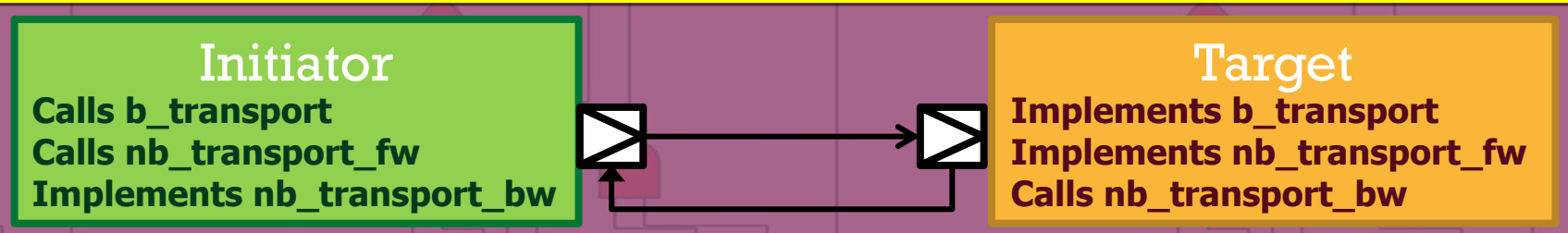
Coding Styles

- 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& );
```

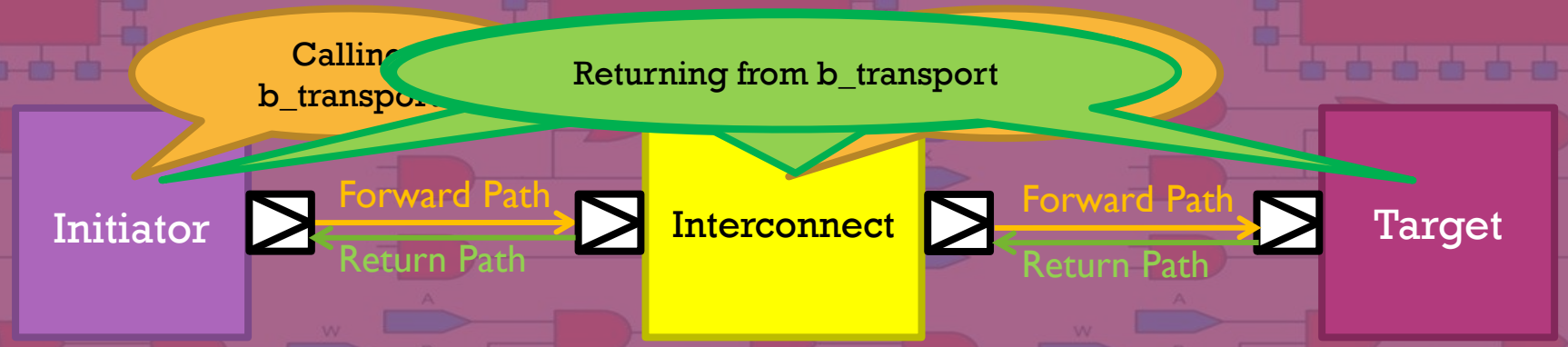
Blocking Transport

- 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

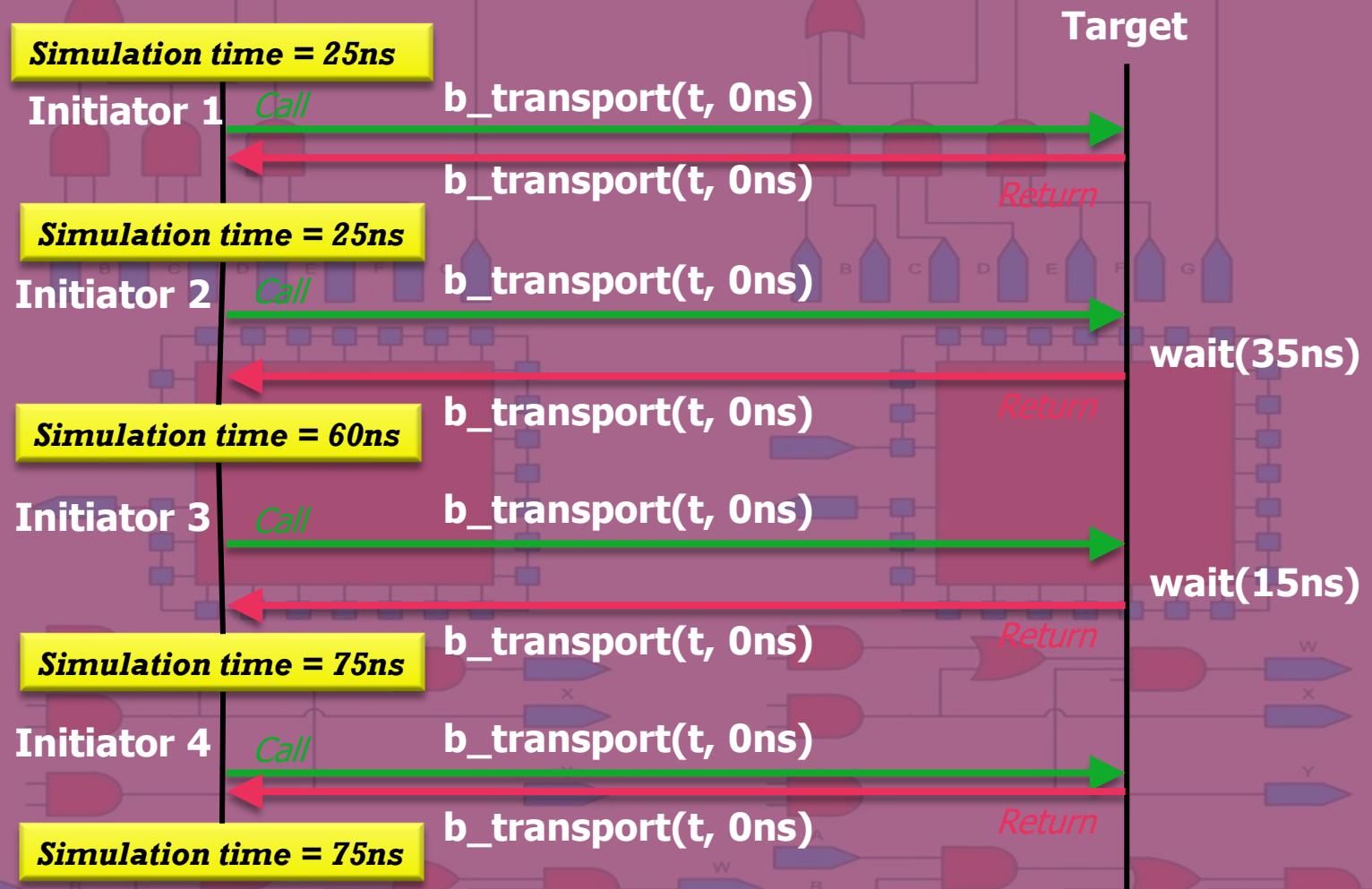
- Forward path
  - Is the calling path by which an initiator or interconnect component makes interface method calls forward in the direction of another interconnect component or the target
- Return path
  - Is the return path by which an initiator, target or interconnect component returns immediately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



Blocking Transport: Path

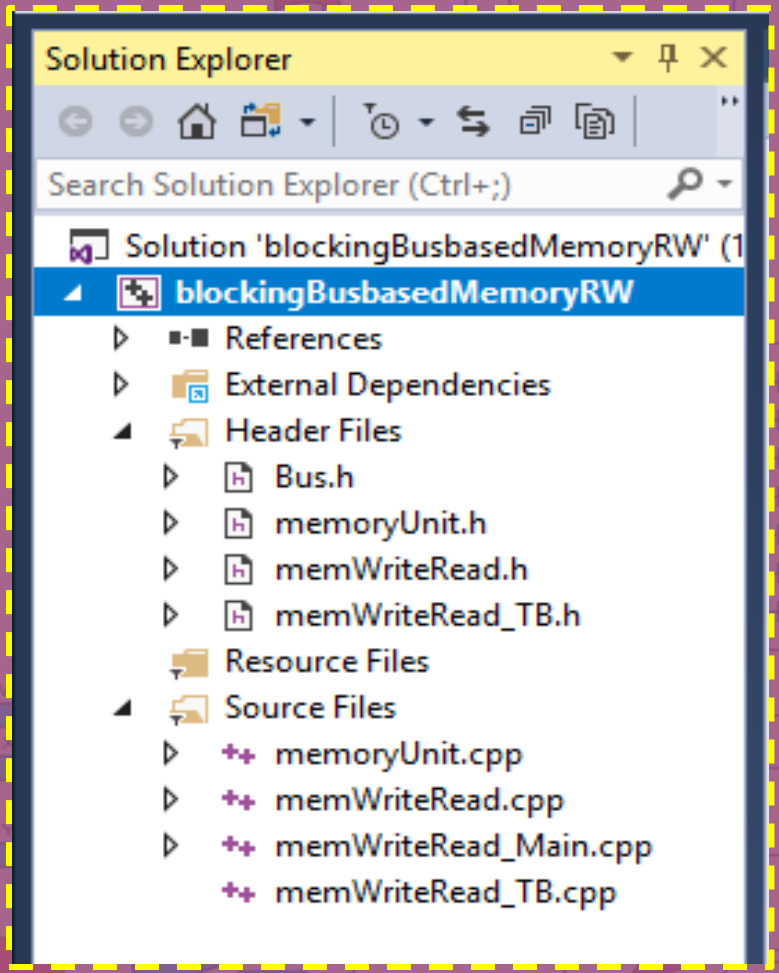
- Timing Diagram

Initiator is blocked until return from b_transport



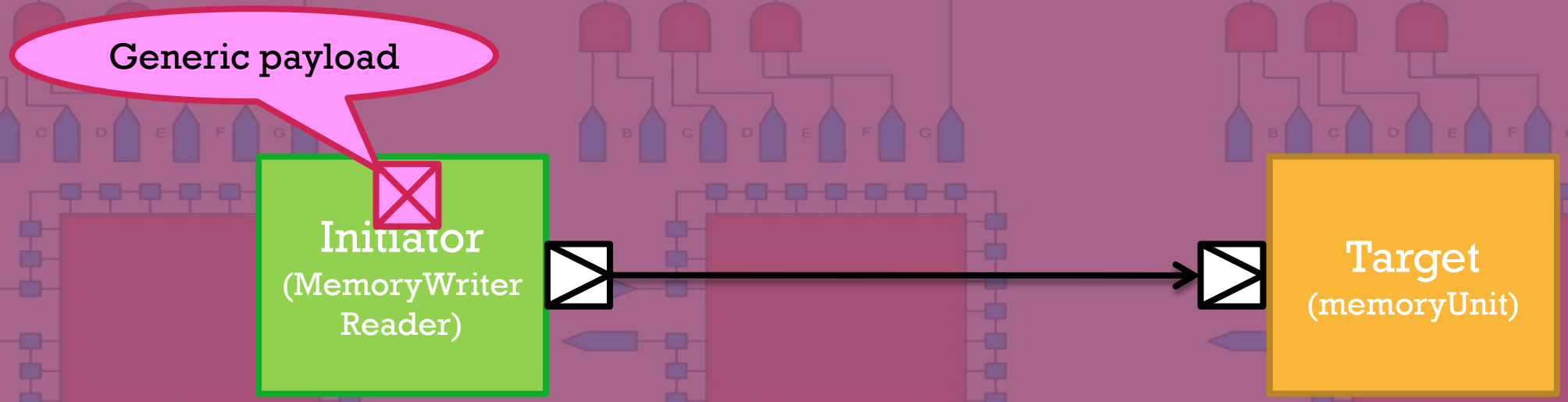
Transaction Level Modeling

- 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

memoryUnit.cpp

- Example 1: blockingBusbasedMemoryRW: Target, *memoryUnit*

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  }
```

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)

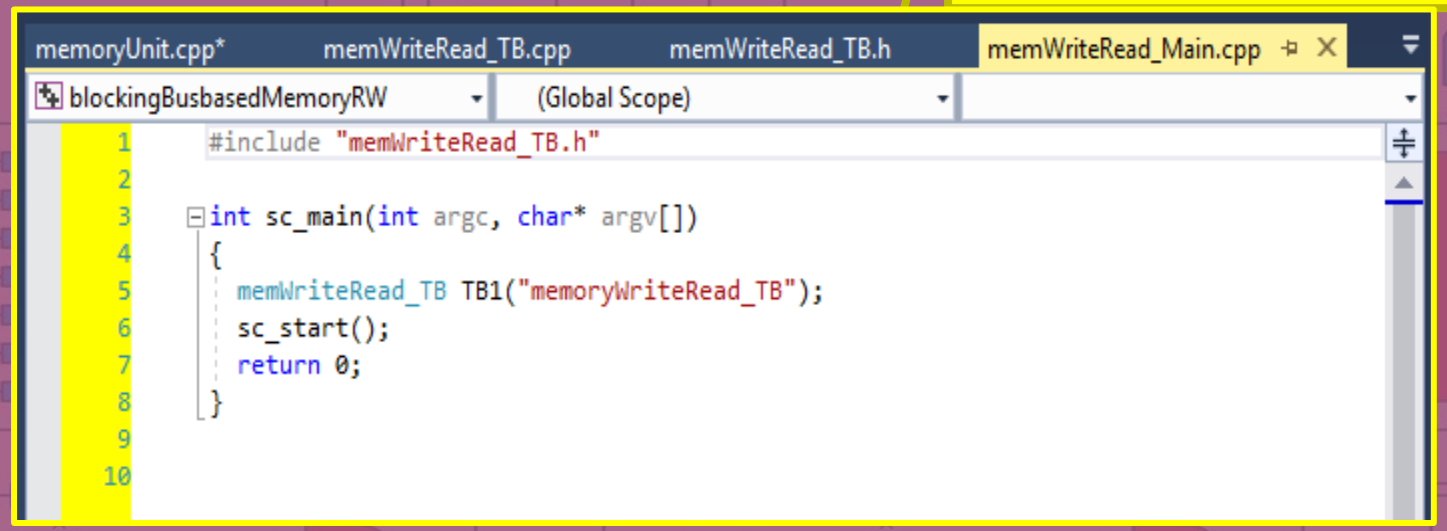
```
1   #include "memWriteRead.h"
2   #include "memoryUnit.h"
3
4   SC_MODULE (memWriteRead_TB) {
5       memWriterReader* MW1;
6       memoryUnit* MU1;
7
8       SC_CTOR(memWriteRead_TB)
9           {
10          MU1 = new memoryUnit("memoryUnit");
11              // MU1->memSideBus(*MU1->membus); // 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



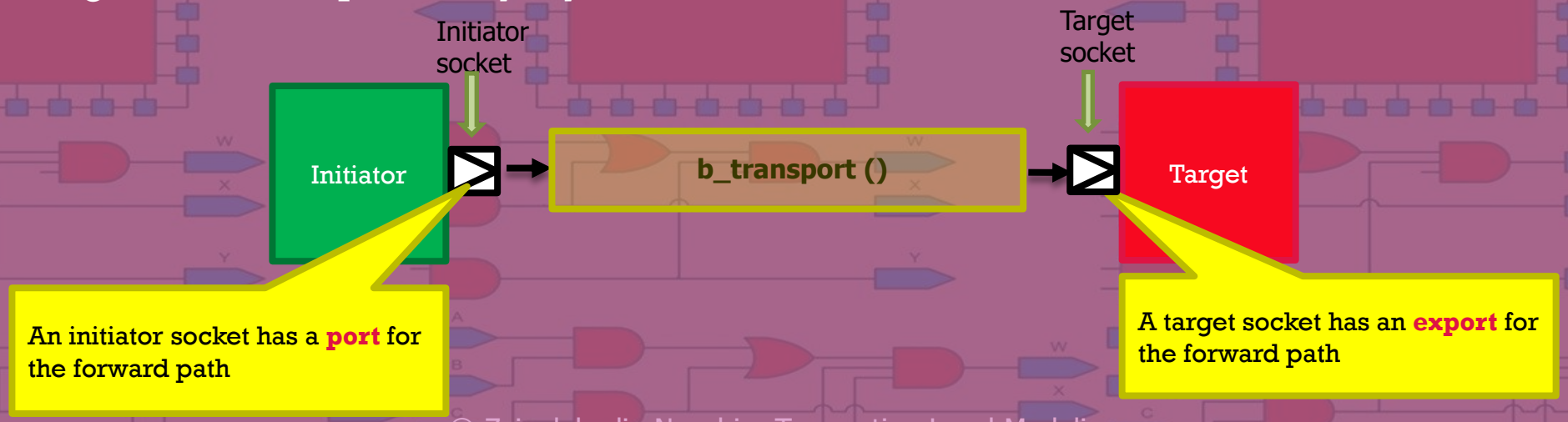
Blocking Transport: Path

- 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.

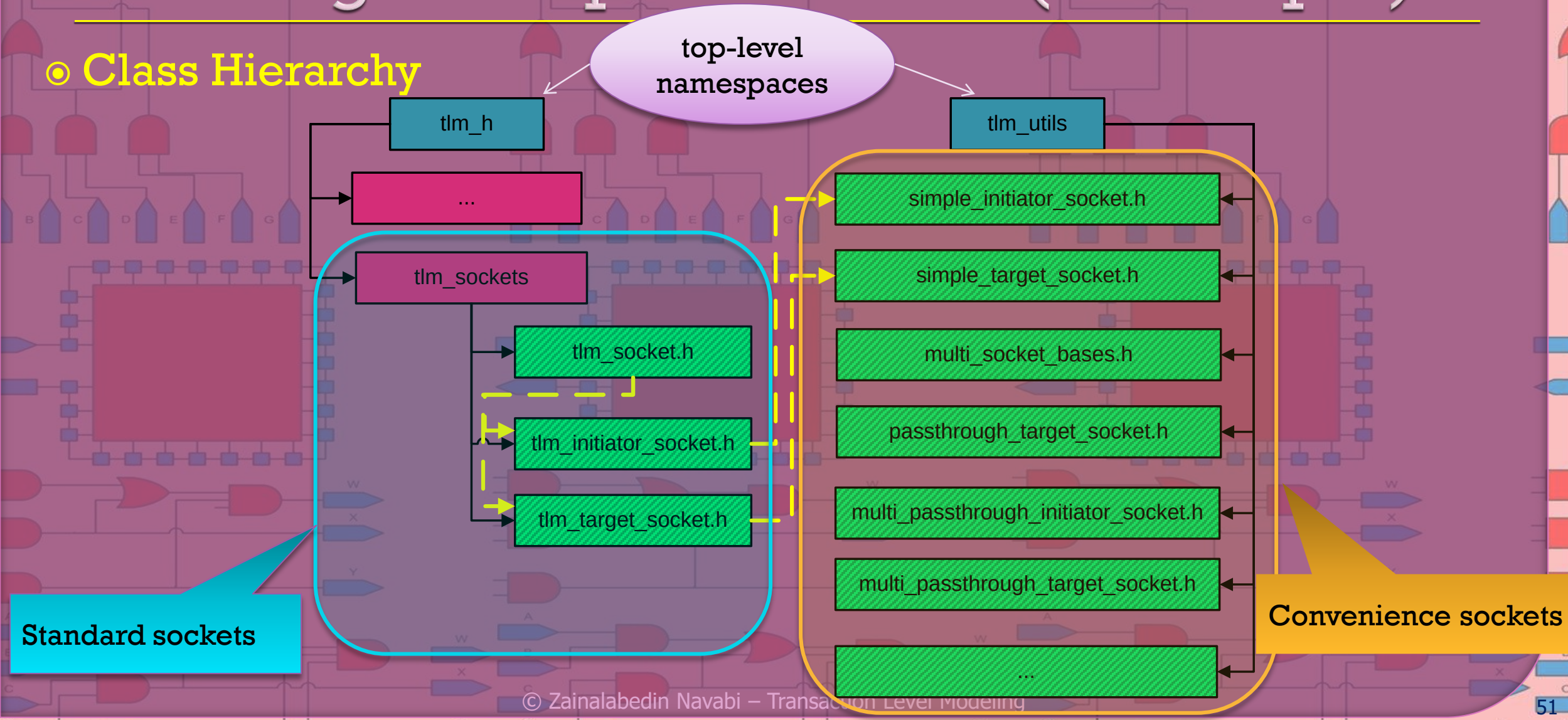
tlm_initiator_socket.h: the conveninence class

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...>
  {
  public:
    tlm_initiator_socket() :
      tlm_base_initiator_socket<BUSWIDTH,
                                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)

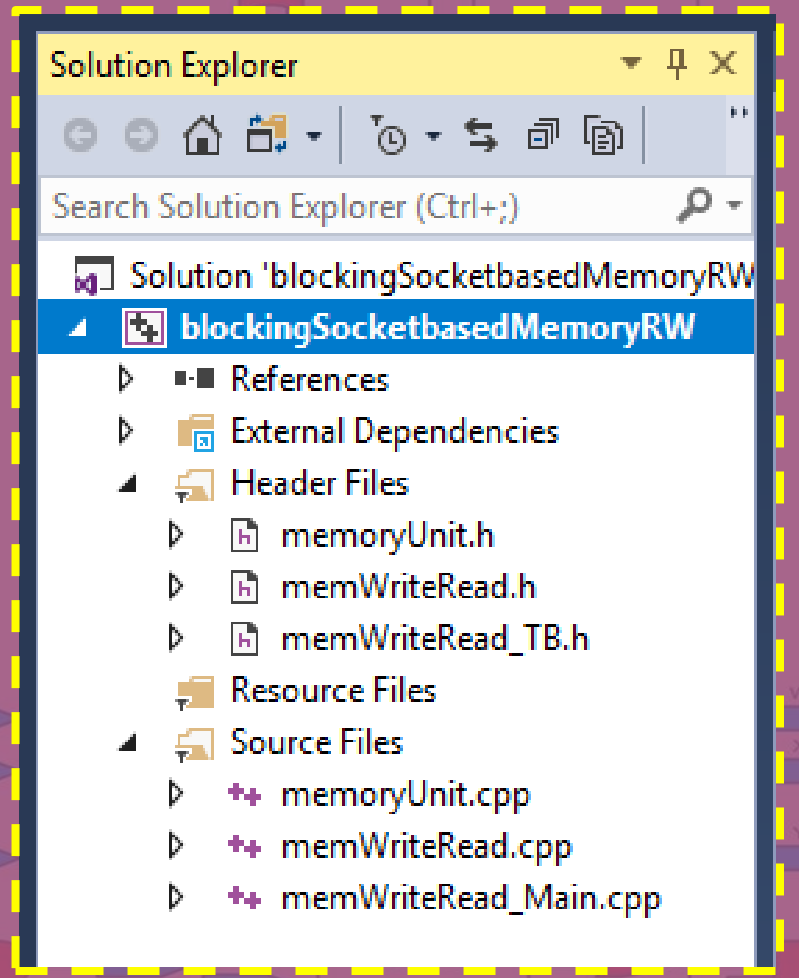
```
70  class process : public tlm::tlm_bw_transport_if<TYPES>
71  {
72  public:
73    typedef sync_enum_type (MODULE::*TransportPtr)(transaction_type&,
74                                                   phase_type&,
75                                                   sc_core::sc_time&);
76    typedef 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:
```

```
113   sync_enum_type nb_transport_bw(transaction_type& trans, phase_type& phase, sc_core::sc_time& t)
114   {
115     if (m_transport_ptr) {
116       // forward call
117       assert(m_mod);
118       return (m_mod->*m_transport_ptr)(trans, phase, t);
119
120     } else {
121       std::stringstream s;
122       s << m_name << ": no transport callback registered";
123       SC_REPORT_ERROR("/OSCI_TLM-2/simple_socket",s.str().c_str());
124     }
125     return tlm::TLM_ACCEPTED;    ///< unreachable code
126   }
147 };
```

- 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)

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

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*

The *b_transport* method may call wait, directly or indirectly

Blocking Transport: Socket (Simple)

- Example 2: blockingSocketbasedMemoryRW: Testbench

memoryWrite_TB.h

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 };
```

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

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

blockingSocketbasedMemoryRW | (Global Scope)

```
1  #include "memWriteRead_TB.h"
2
3  int sc_main(int argc, char* argv[])
4  {
5    memWriteRead_TB TB1("memoryWriteRead_TB");
6    sc_start();
7    return 0;
8  }
```

Blocking Transport: Socket (Simple)

- Example 2: blockingSocketbasedMemoryRW: Output

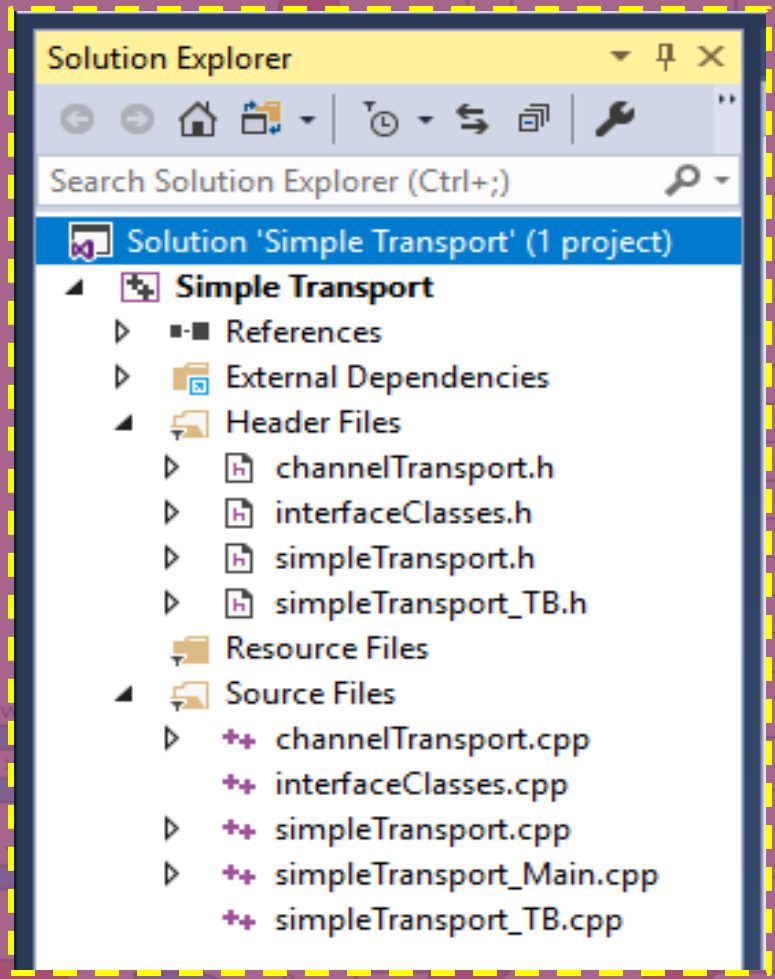
```
        SystemC 2.3.2-Accellera --- Nov 16 2018 05:36:55
        Copyright (c) 1996-2017 by all Contributors,
        ALL RIGHTS RESERVED
At time: 13 nsWR: W, Iteration:0 data:00000101 00000110 00000111 00001000 00001001
At time: 26 nsWR: W, Iteration:11 data:00010000 00010001 00010010 00010011 00010100
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

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

Simple Transport | (Global Scope)

```
1   #include "simpleTransport.h"
2
3   SC_MODULE (simpleTransport_TB) {
4
5       transmitter* TRS1;
6       receiver* RCV1;
7
8       SC_CTOR(simpleTransport_TB) {
9           RCV1 = new receiver("receiver");
10          TRS1 = new transmitter("transmitter");
11              TRS1->out(*RCV1->insideTargetChannel);
12      }
13  };
14
```

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 timin
  - 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)

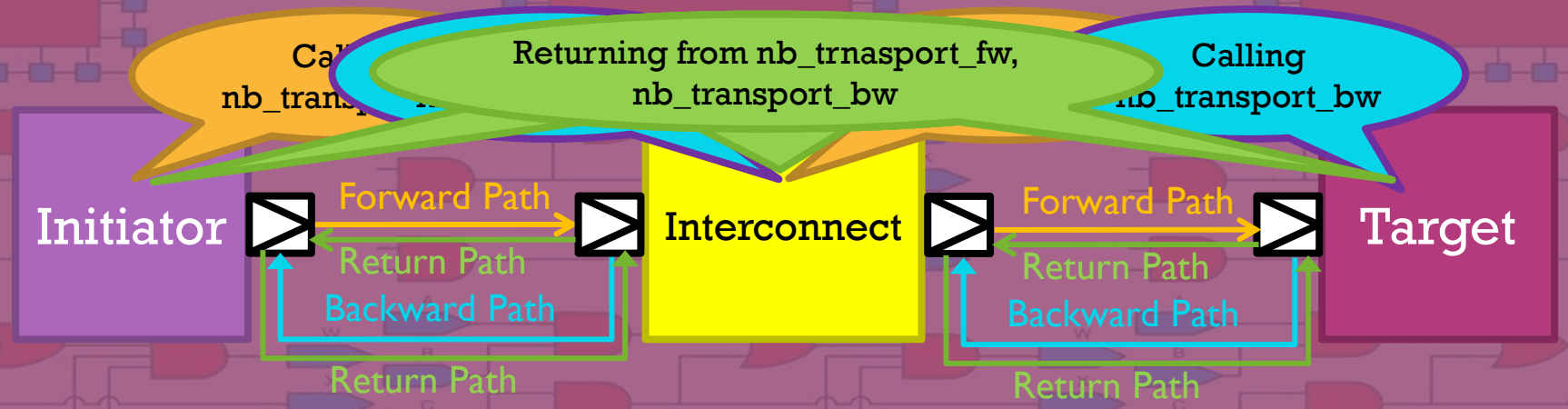
TLM_UPDATED
- Transaction, phase and timing arguments updated (used) on return
- Target has advanced the protocol state machine to the next state

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



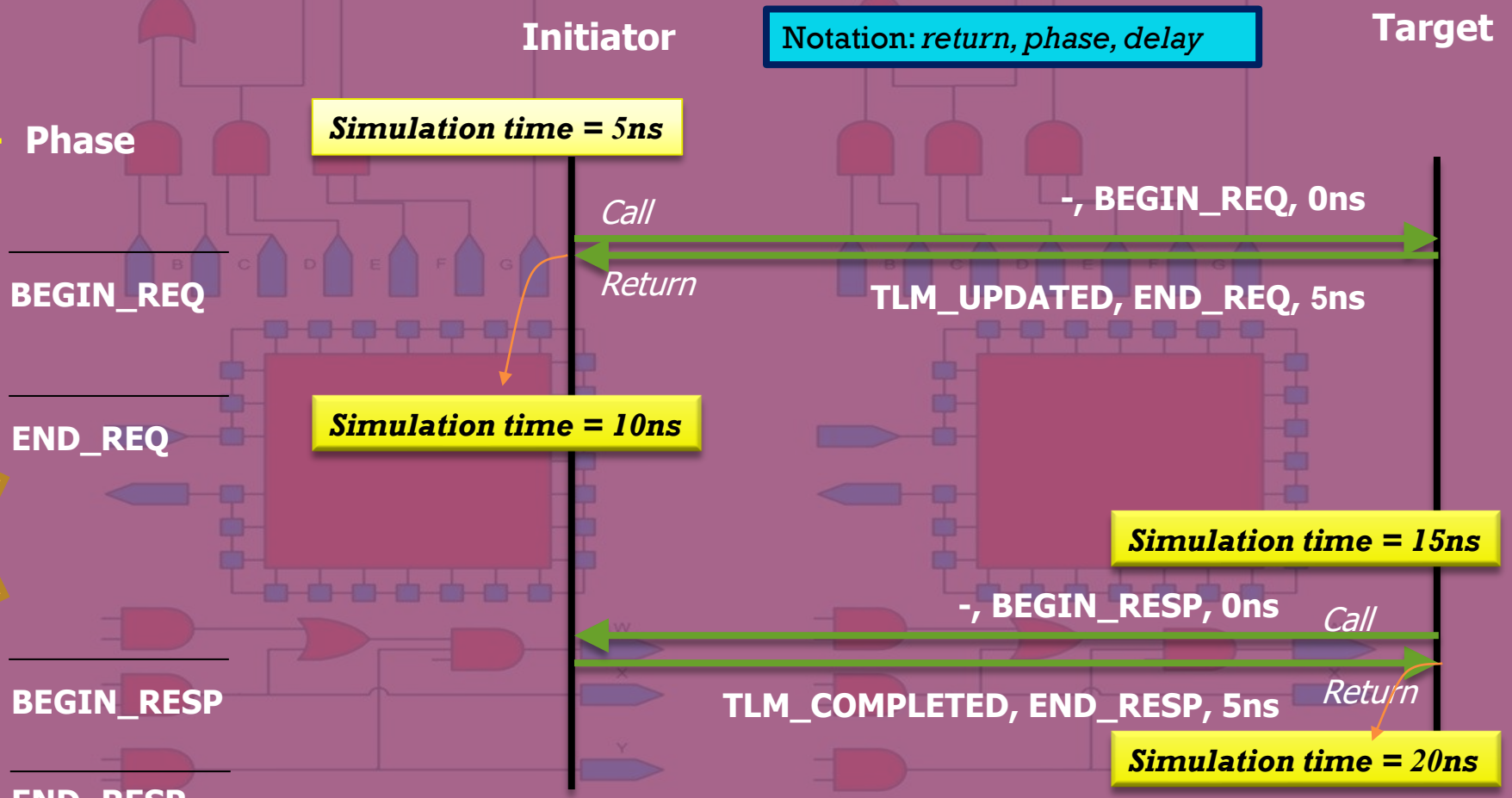
Non-blocking Transport: Path

- Timing Diagram: Using The Backward Path

Transaction accepted now, caller asked to wait

Non-blocking Transport: Path

- Timing Diagram: Using The Return Path



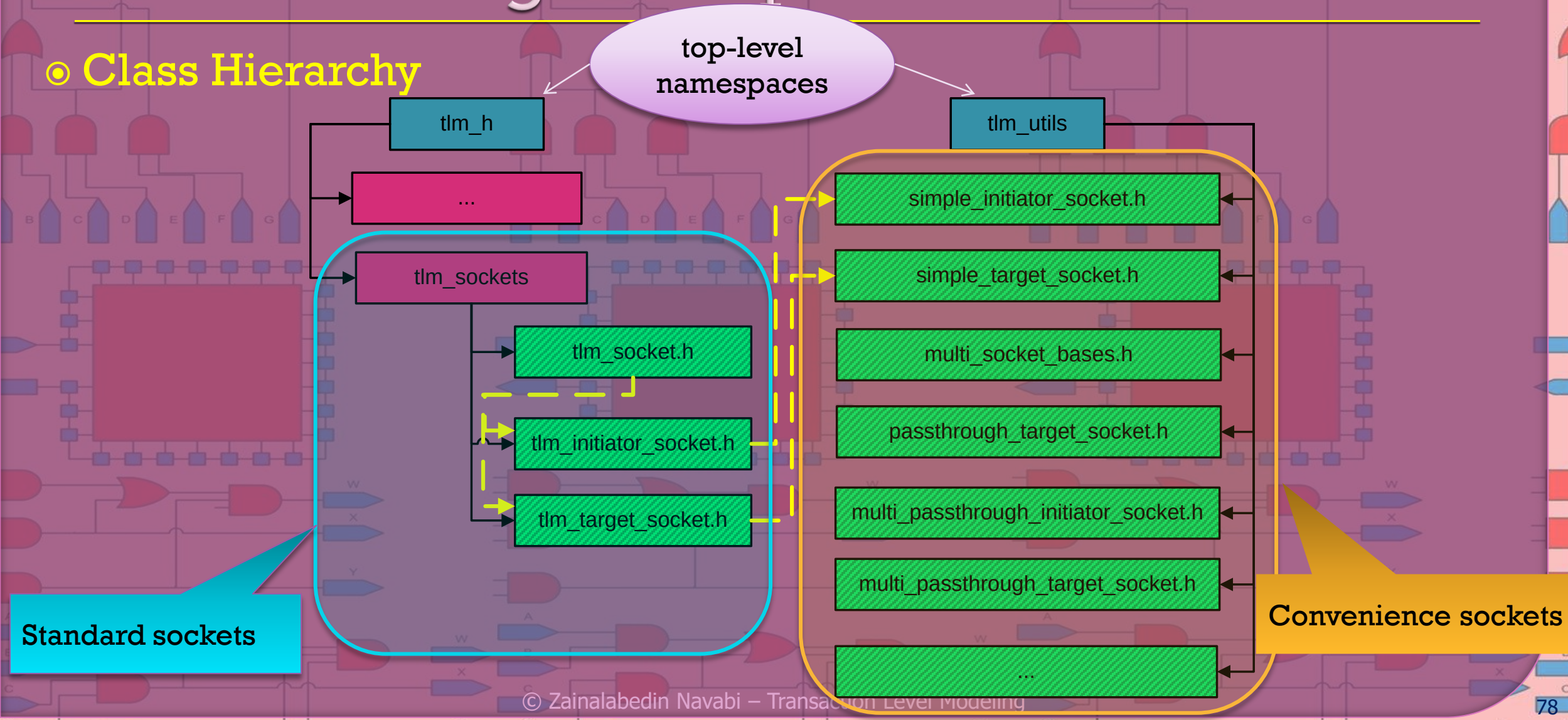
Callee annotates delay to next transition, caller waits

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

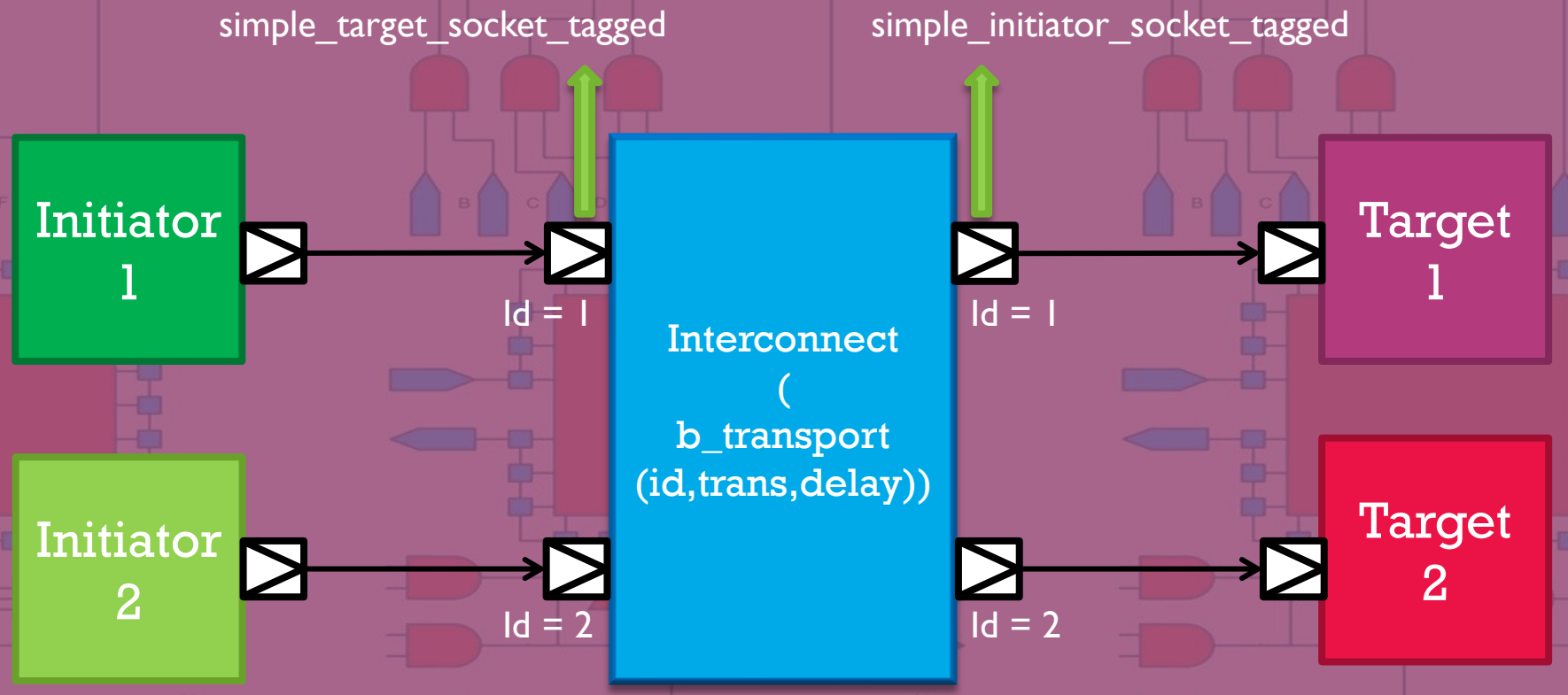
```
151 template <typename MODULE,
152           unsigned int BUSWIDTH = 32,
153           typename TYPES = tlm::tlm_base_protocol_types>
154 class simple_initiator_socket_tagged :
155   public tlm::tlm_initiator_socket<BUSWIDTH, TYPES>
156 {
157 public:
158   typedef typename TYPES::tlm_payload_type             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:



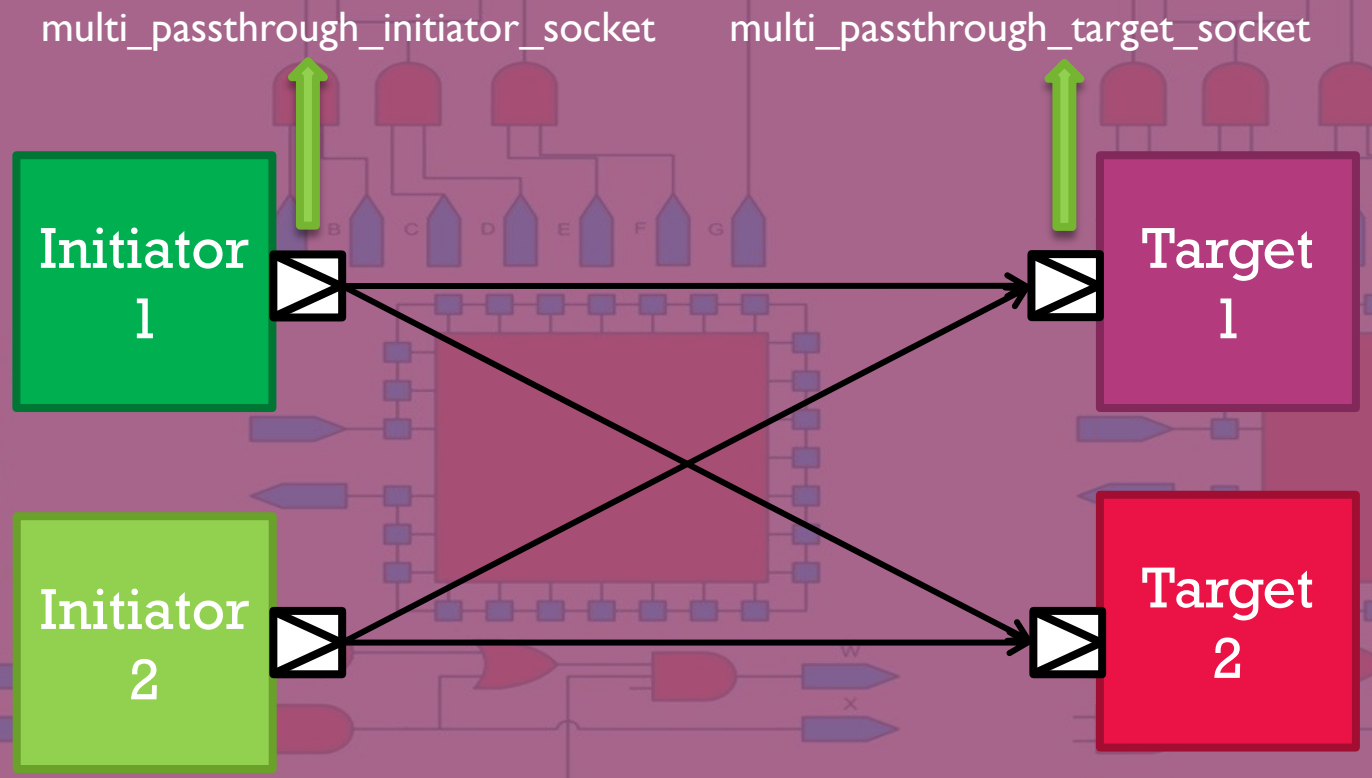
Distinguish origin of incoming transactions using socket *id*

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

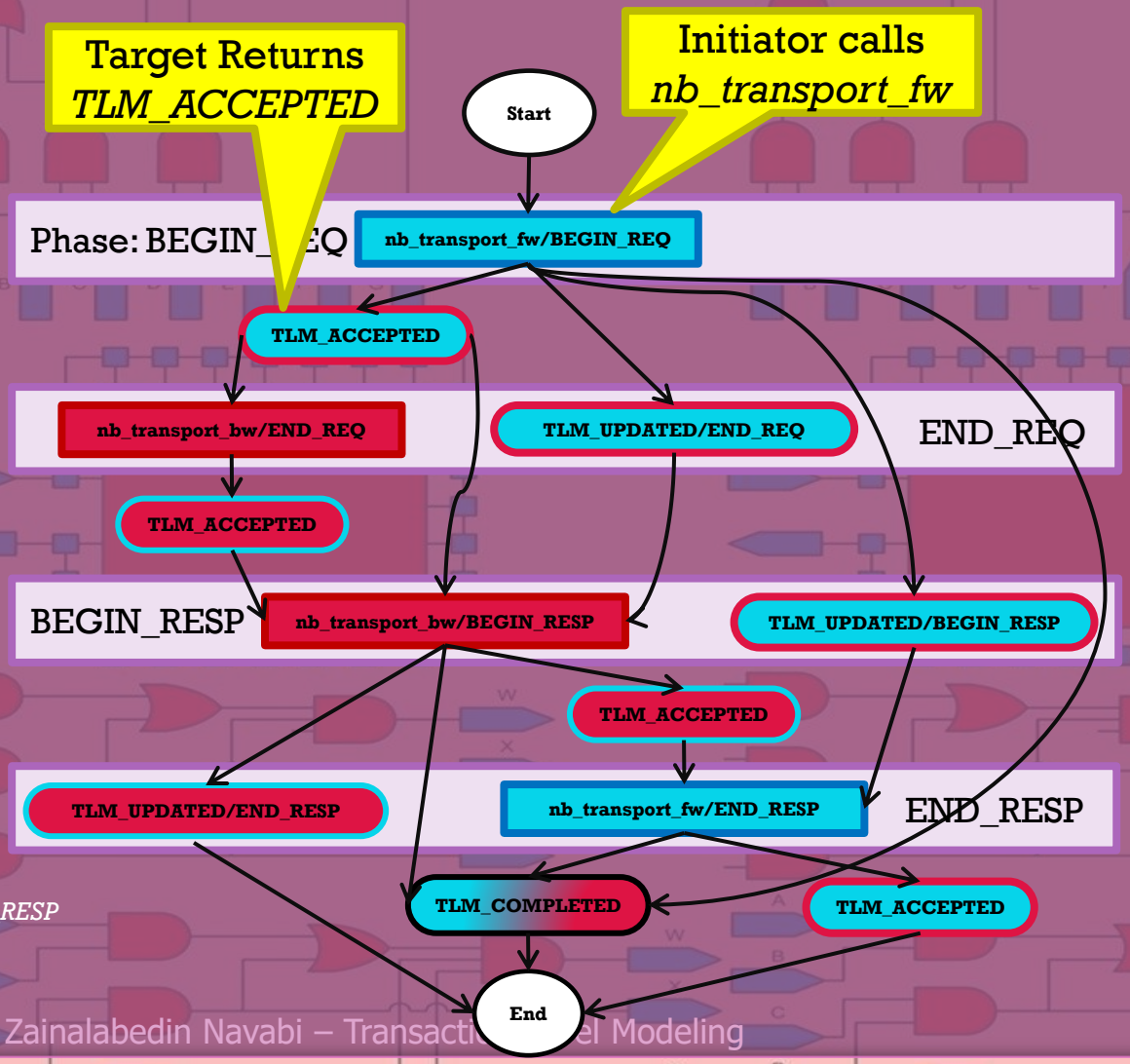
- The base protocol = tlm_generic_payload + tlm_phase
- consists of a set of rules to ensure maximal interoperability between transaction level models of components

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

For base protocol scenarios, there is not necessarily a unique hardware implementation, however, for every scenario, there is always some hardware correspondences

Non-blocking Transport: Phase & Base Protocol

- The Initiator: Calls nb_transport_fw
- The Target
  - Executes nb_transport_fw
  - Modifies the phase and put the transaction in the peq
  - Return *TLM_ACCEPTED* to the initiator
- The Initiator: Waits for receiving *END_REQ*
- In the Target:
  - Transaction is expired
  - *Peq_cb* is executed: Calls nb_transport_bw
- The Initiator:
  - Executes nb_transport_bw
  - Modify the phase and put the transaction in the peq
  - Return *TLM_ACCEPTED* to the target
- The Target:
  - Put the transaction with *BEGIN_RESP* in the peq
  - Waits for expiration of the transaction
    - *Peq_cb* is executed: Calls nb_transport_bw to send *BEGIN_RESP*
- The Initiator:
  - Executes nb_transport_bw
  - Modify the phase and put the transaction in the peq
  - Return *TLM_ACCEPTED* to the target
  - Waits for expiration of the transaction
    - *Peq_cb* is executed: Calls nb_transport_fw to send *END_RESP*
- The Target
  - Executes nb_transport_fw
  - Return *TLM_ACCEPTED* to the initiator



Nomenclature

Blue: Initiator
Red: Target

Call:

nb_transport(GP, phase, delay)

Return path from target to initiator:

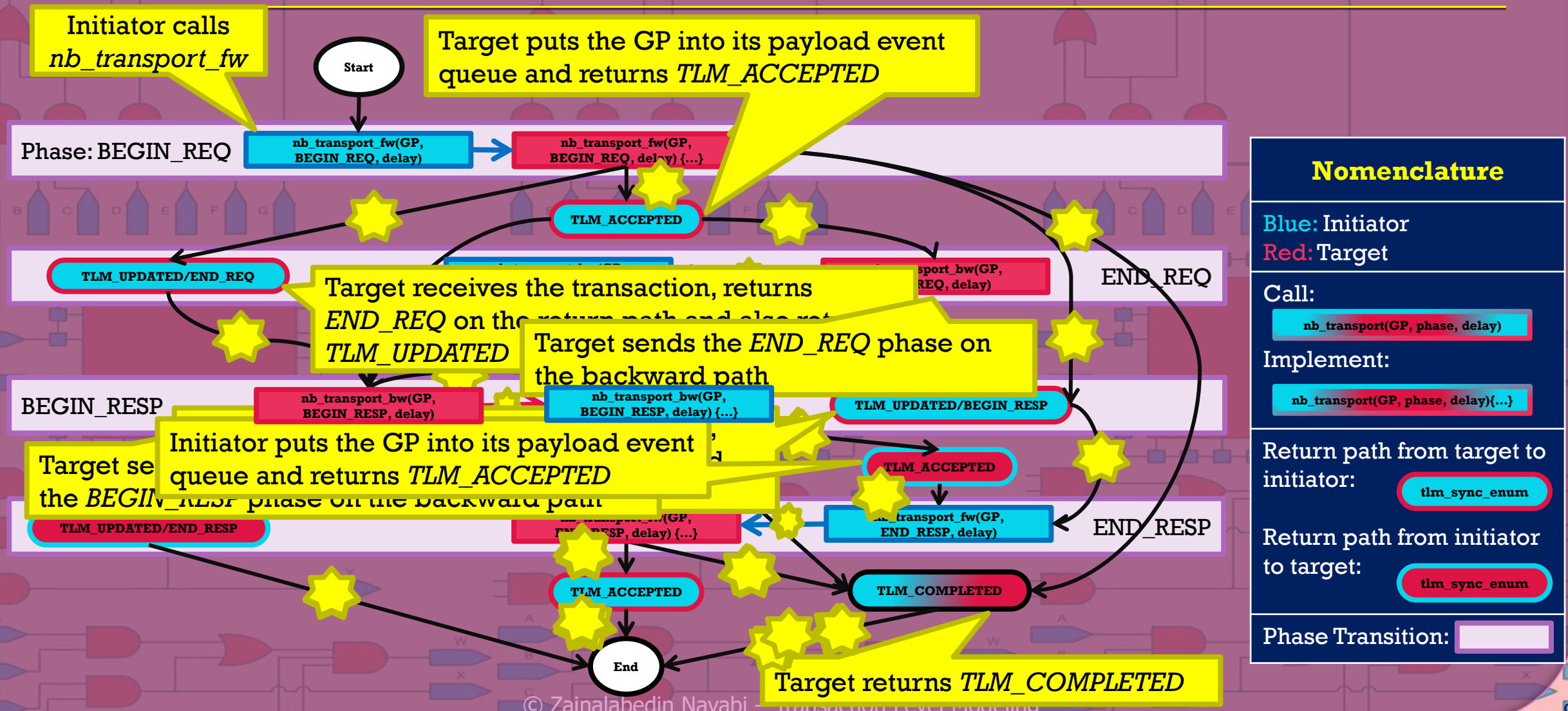
tlm_sync_enum

Return path from initiator to target:

tlm_sync_enum

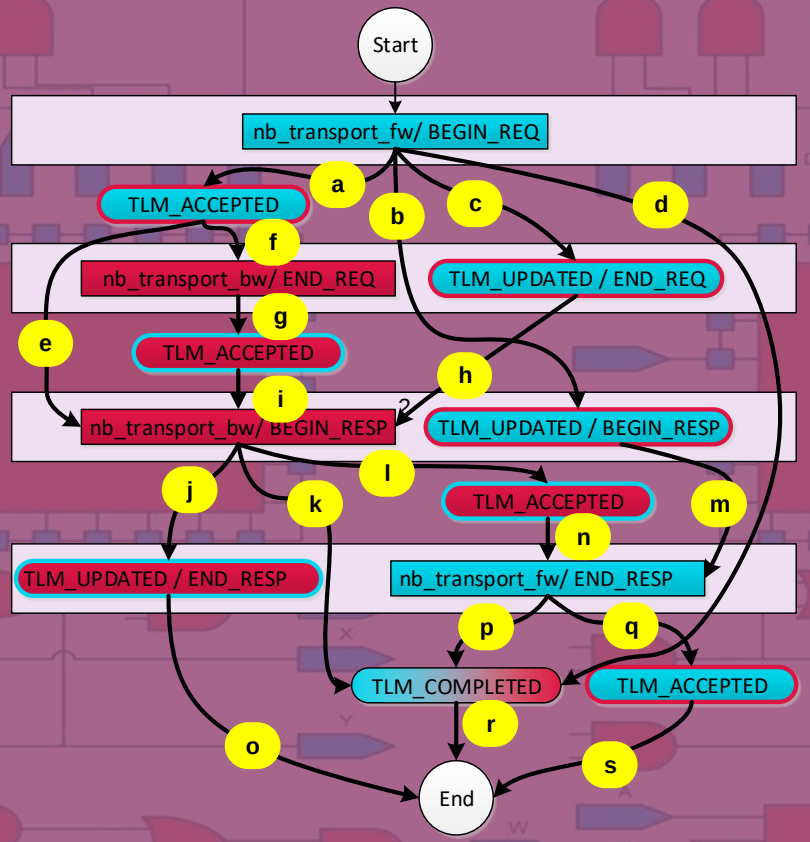
Phase Transition:

Non-blocking Transport: Phase & Base Protocol



Non-blocking Transport: Phase & Base Protocol

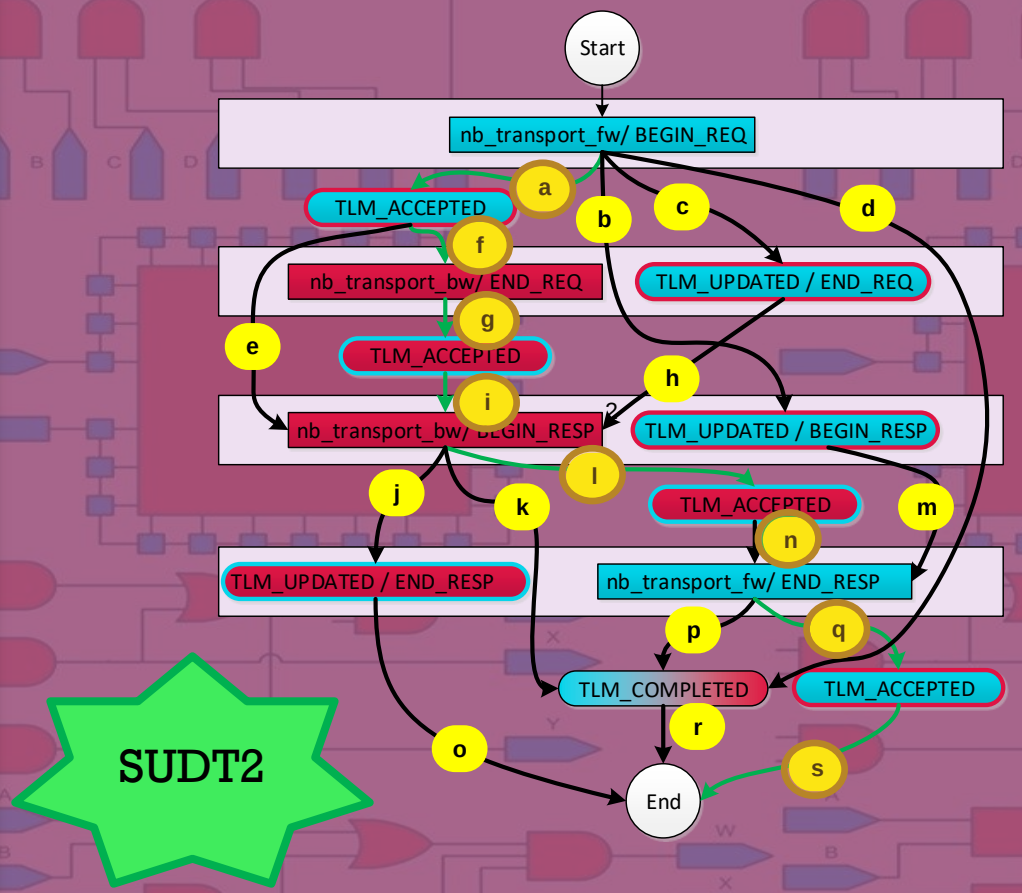
◉ Abstract Communication All Scenarios



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

◉ Abstract Communication SUDT2 Scenario

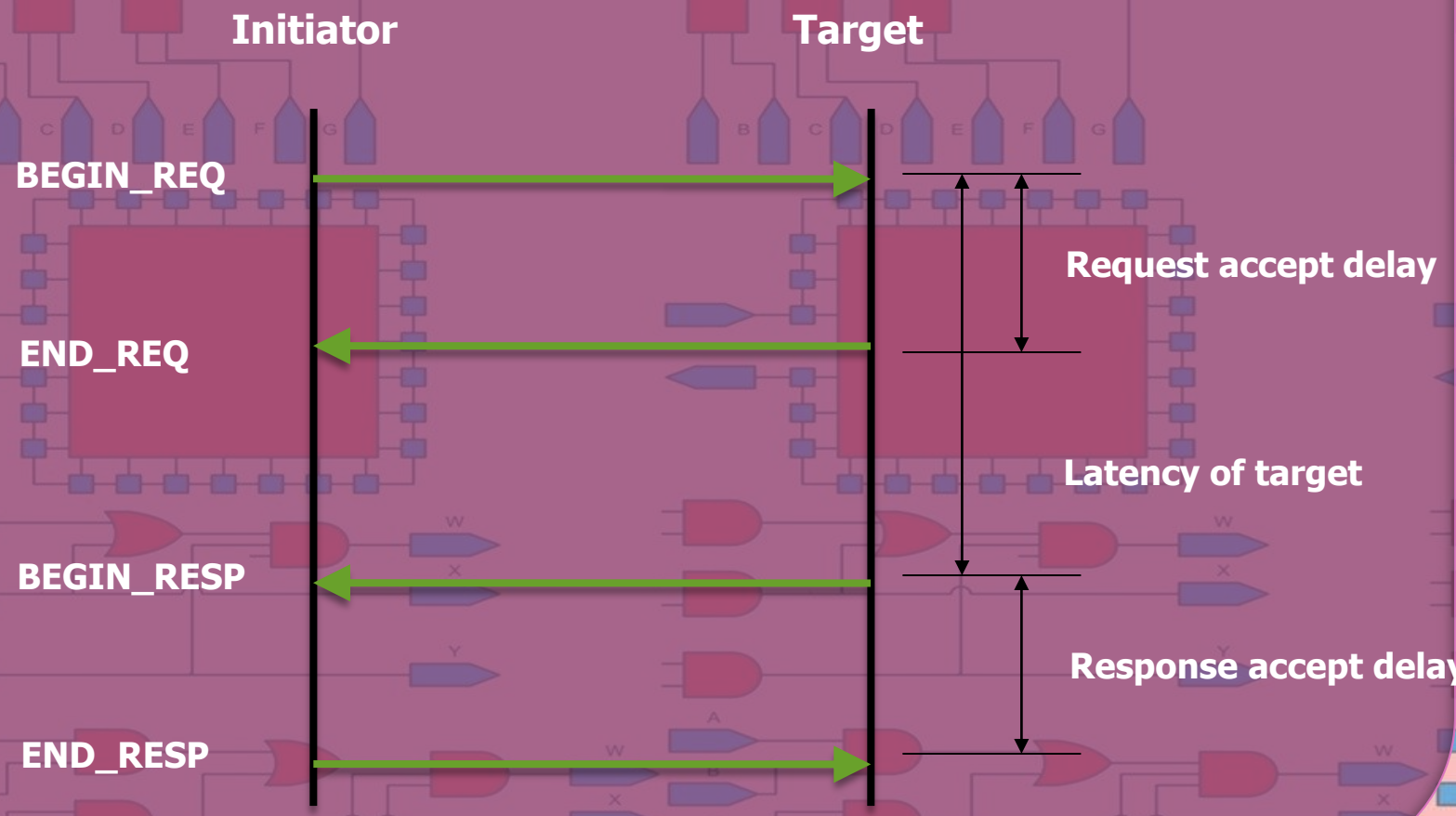


TLM Comm. Scenario	TLM-2.0 base protocol flow: Edge id.							
SUDT2	c	h	j	o				
	a	f	g	i	j	o		
	c	h	l	n	q	s		
	a	f	g	i	l	n	q	s
SUDT3	a	e	j	o				
	a	e	l	n	q	s		
SUDT4	a	e	l	n	q	s		
	a	e	l	n	p	r		
	b	m	q	s				
	b	m	p	r				
SUDT5	d	r						
SUDT7	c	h	k	r				
	a	f	g	i	k	r		
SUDT8	a	e	k	r				
	a	e	l	n	q	s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

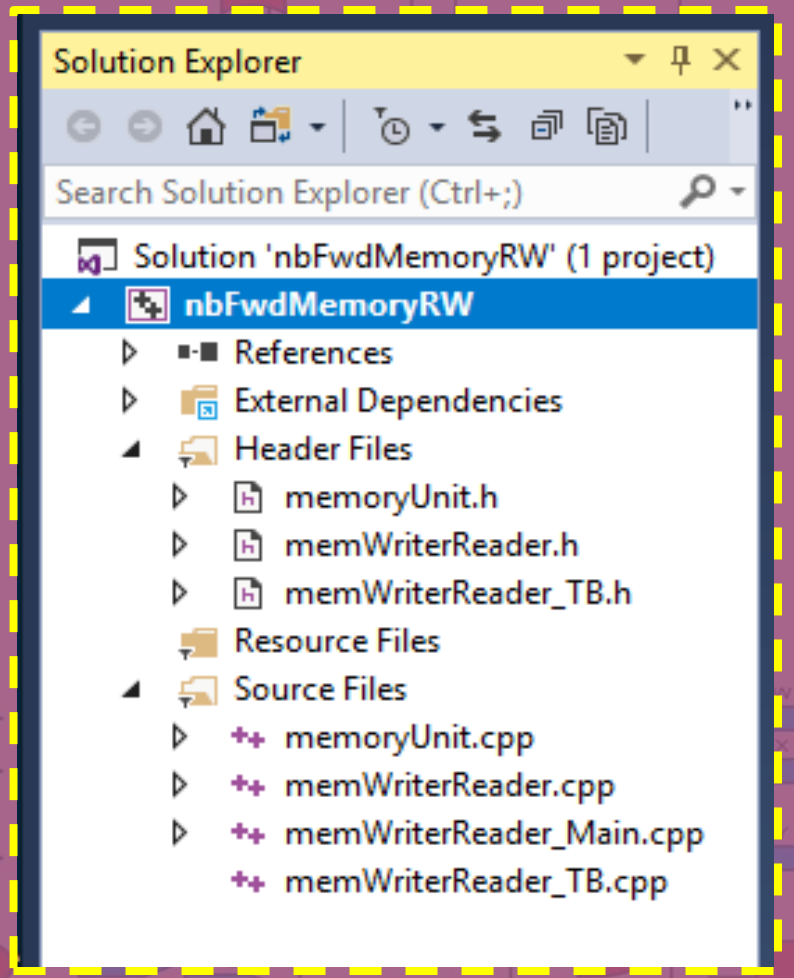
⊙ Base Protocol Timing Parameters and Flow Control

BEGIN_REQ must wait for previous END_REQ, BEGIN_RESP for END_RESP



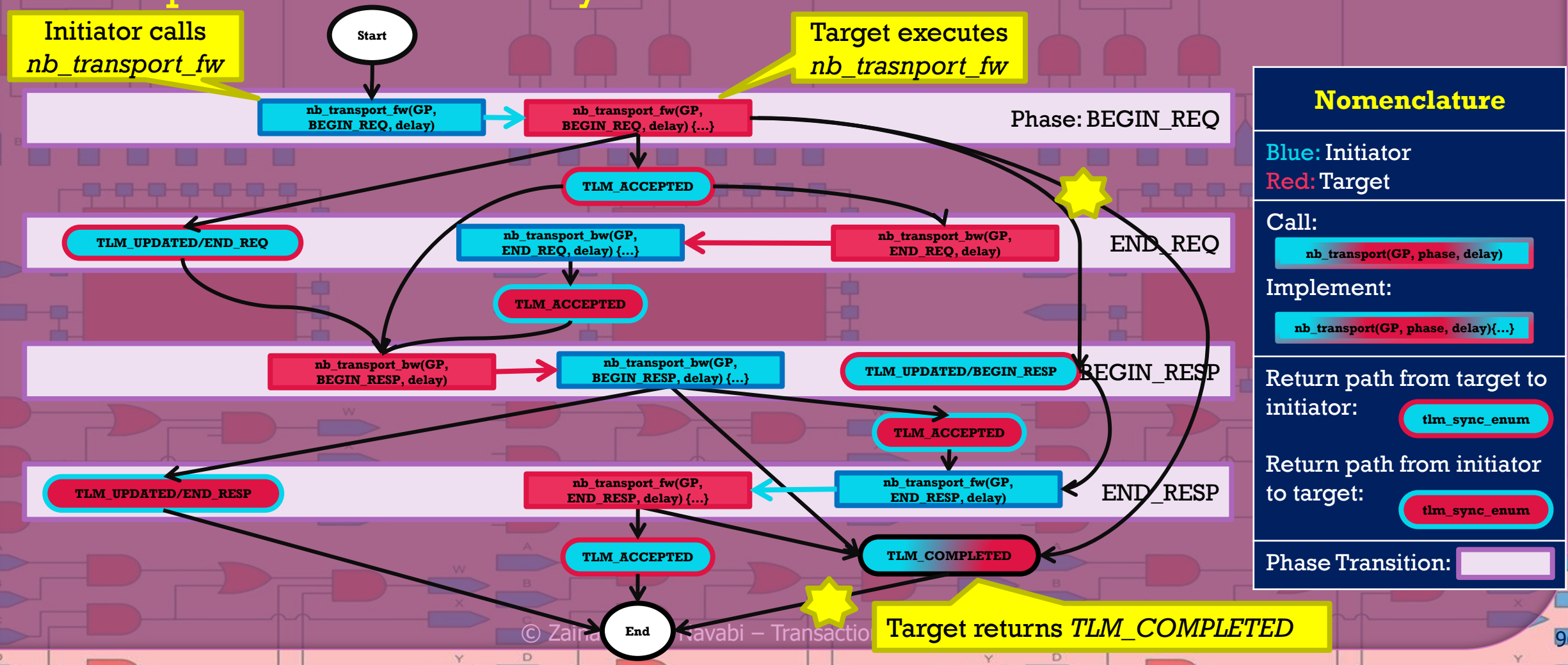
Transaction Level Modeling

- 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



Nomenclature

Blue: Initiator
Red: Target

Call:
`nb_transport(GP, phase, delay)`

Implement:
`nb_transport(GP, phase, delay){...}`

Return path from target to initiator:
`tlm_sync_enum`

Return path from initiator to target:
`tlm_sync_enum`

Phase Transition:

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

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

nbFwdMemoryRW | (Global Scope)

```
43
    void memWriterReader::doSomethingGood( tlm::tlm_generic_payload& completeTrans,
                                           sc_time totalTime )
46  {
47      if ( completeTrans.is_response_error() )
48          SC_REPORT_ERROR("TLM-2", "error...\n");
49
50      tlm::tlm_command cmd = completeTrans.get_command();
51      uint64           adr = completeTrans.get_address();
52      int*             ptr = reinterpret_cast<int*>( completeTrans.get_data_ptr() );
53
54      cout << "Above was completed @time " << totalTime << '\n';
55  }
```

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)
 1  #include "memoryUnit.h"
 2
 3  tlm::tlm_sync_enum memoryUnit::nb_transport_fw
 4        (tlm::tlm_generic_payload& receivedTrans,
 5         tlm::tlm_phase& phase, sc_time& delay ){
 6
 7     tlm::tlm_command cmd = receivedTrans.get_command();
 8     uint64           adr = receivedTrans.get_address();
 9     unsigned char*   ptr = receivedTrans.get_data_ptr();
10     unsigned int     len = receivedTrans.get_data_length();
11     unsigned char*   byt = receivedTrans.get_byte_enable_ptr();
12     unsigned int     wid = receivedTrans.get_streaming_width();
13
14     if (byt != 0) {
15        receivedTrans.set_response_status( tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE );
16        return tlm::TLM_COMPLETED;
17     }
18     if (len > 5 || wid < len) {
19        receivedTrans.set_response_status( tlm::TLM_BURST_ERROR_RESPONSE );
20        return tlm::TLM_COMPLETED;
21     }
22     unsigned int i;
23     if ( cmd == tlm::TLM_READ_COMMAND )
24        for(i=0; i<len; i=i+1) {
25           *(ptr+i) = *((unsigned char*) (memArray+adr+i));
26        }
27     else if ( cmd == tlm::TLM_WRITE_COMMAND )
28        for(i=0; i<len; i=i+1) {
29           *((unsigned char*) (memArray+adr+i)) = *(ptr+i);
30        }
31
32     receivedTrans.set_response_status( tlm::TLM_OK_RESPONSE );
33     delay = delay + sc_time(123, SC_NS);
34     return tlm::TLM_COMPLETED;
35  }
```

- Example 4: nbFwdMemoryRW: Target, *memoryUnit*

The *nb_transport* methods shall not call wait, directly or indirectly

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 };
```

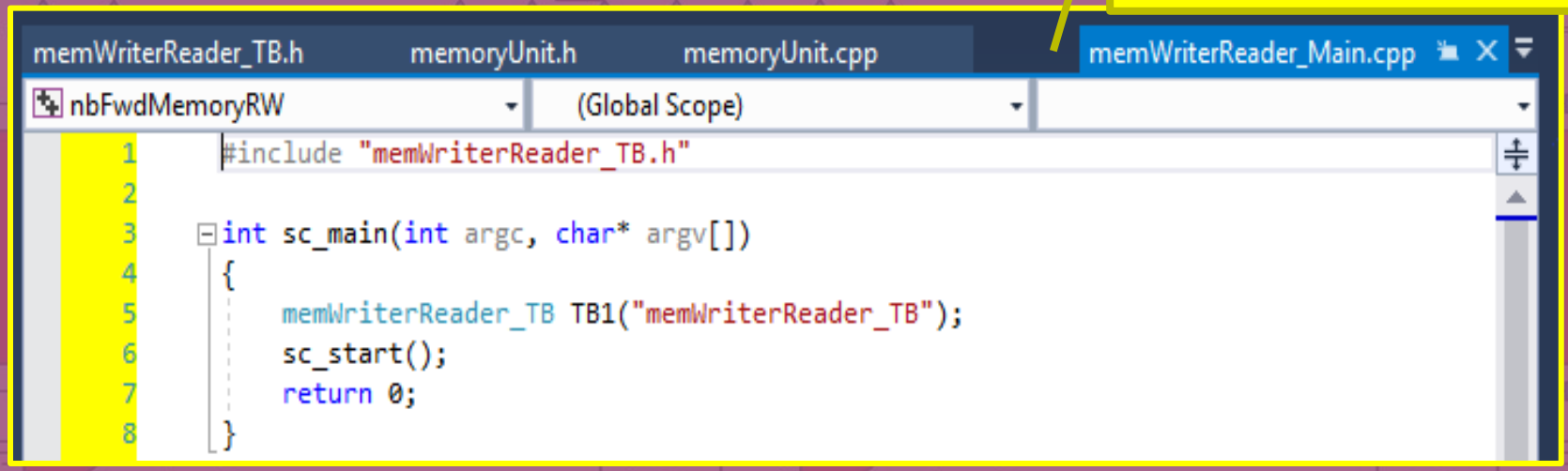
Instantiate components

- 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



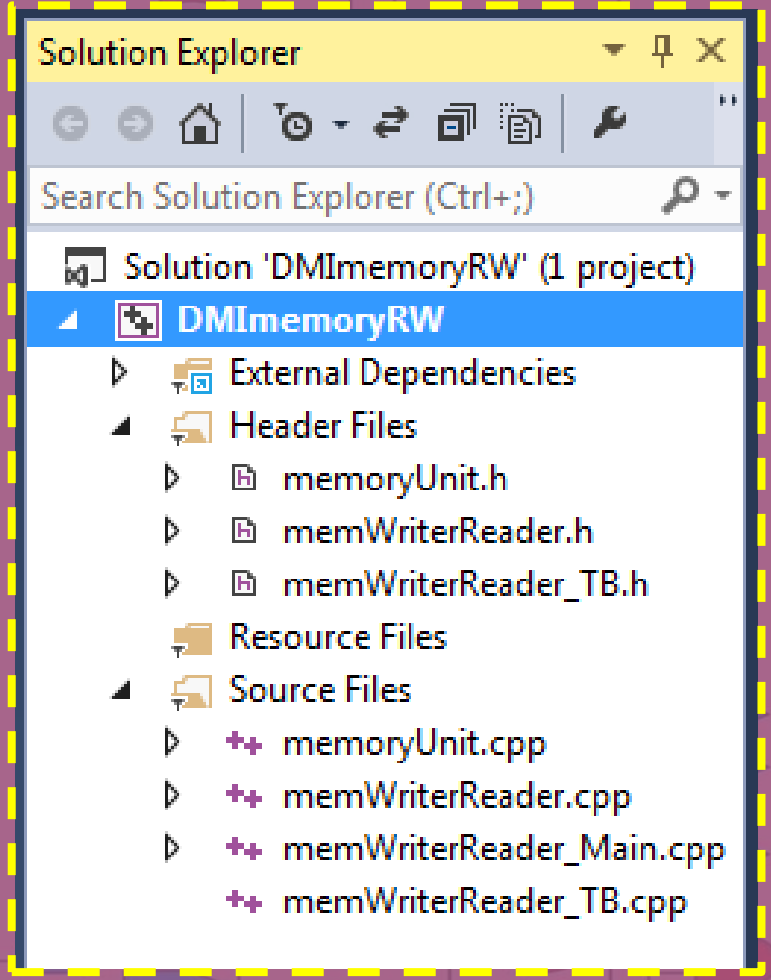
Non-blocking Transport: Forward

- 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
```

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
      - DMImemoryRW Example
- + Complete System



DMI Transport

- Direct Memory Interface (DMI) provides a direct access to an area of memory for an initiator
- The DMI offers a large increase in simulation speed
- DMI uses both forward and backward interface
  - The forward DMI interface is used to request a DMI access (e.g., read, write or both) to a given address. *get_direct_mem_ptr* interface method is called on the forward path from initiator to target
  - The backward DMI interface is used to invalidate DMI pointers previously established using the forward path. *invalidate_direct_mem_ptr* is called on the backward path from target to initiator
- The DMI interfaces use DMI transaction object, *tlm_dmi*
- Only makes sense with loosely-timed models

DMI Transaction Object

tlm_dmi.h

tlm::tlm_dmi

```
30  enum dmi_access_e
31  { DMI_ACCESS_NONE       = 0x00
32  , DMI_ACCESS_READ       = 0x01
33  , DMI_ACCESS_WRITE      = 0x02
34  , DMI_ACCESS_READ_WRITE = DMI_ACCESS_READ | DMI_ACCESS_WRITE
35  };
```

tlm_dmi.h

tlm::tlm_dmi

```
37  tlm_dmi (void)
38  {
39    init();
40  }
41
42  void init (void)
43  {
44    m_dmi_ptr           = 0x0;
45    m_dmi_start_address = 0x0;
46    m_dmi_end_address   = (sc_dt::uint64)(-1);
47    m_dmi_access        = DMI_ACCESS_NONE;
48    m_dmi_read_latency  = sc_core::SC_ZERO_TIME;
49    m_dmi_write_latency = sc_core::SC_ZERO_TIME;
50  }
```

Granted access

Define the latency of read/write transactions

tlm_dmi.h

(Global Scope)

Tlm_dmi.h

```
27  class tlm_dmi {
28    public:
29
30    enum dmi_access_e { ... };
36
      tlm_dmi (void) { ... }
      void init (void) { ... }
42
51
```

Different access types for indicating the access granted to the initiator

Invalidate all attributes

```
unsigned char*    get_dmi_ptr          (void) const {return m_dmi_ptr;}
sc_dt::uint64     get_start_address    (void) const {return m_dmi_start_address;}
sc_dt::uint64     get_end_address      (void) const {return m_dmi_end_address;}
sc_core::sc_time  get_read_latency     (void) const {return m_dmi_read_latency;}
sc_core::sc_time  get_write_latency    (void) const {return m_dmi_write_latency;}
dmi_access_e      get_granted_access   (void) const {return m_dmi_access;}
bool              is_none_allowed      (void) const {return m_dmi_access == DMI_ACCESS_NONE;}
bool              is_read_allowed      (void) const {return (m_dmi_access & DMI_ACCESS_READ) == DMI_ACCESS_READ;}
bool              is_write_allowed     (void) const {return (m_dmi_access & DMI_ACCESS_WRITE) == DMI_ACCESS_WRITE;}
bool              is_read_write_allowed (void) const {return (m_dmi_access & DMI_ACCESS_READ_WRITE) == DMI_ACCESS_READ_WRITE;}
```

The attributes are used by target

```
void              set_dmi_ptr          (unsigned char* p)   {m_dmi_ptr = p;}
void              set_start_address    (sc_dt::uint64 addr) {m_dmi_start_address = addr;}
void              set_end_address      (sc_dt::uint64 addr) {m_dmi_end_address = addr;}
void              set_read_latency     (sc_core::sc_time t) {m_dmi_read_latency = t;}
void              set_write_latency    (sc_core::sc_time t) {m_dmi_write_latency = t;}
void              set_granted_access   (dmi_access_e a)     {m_dmi_access = a;}
void              allow_none           (void)               {m_dmi_access = DMI_ACCESS_NONE;}
void              allow_read           (void)               {m_dmi_access = DMI_ACCESS_READ;}
void              allow_write          (void)               {m_dmi_access = DMI_ACCESS_WRITE;}
void              allow_read_write     (void)               {m_dmi_access = DMI_ACCESS_READ_WRITE;}
```

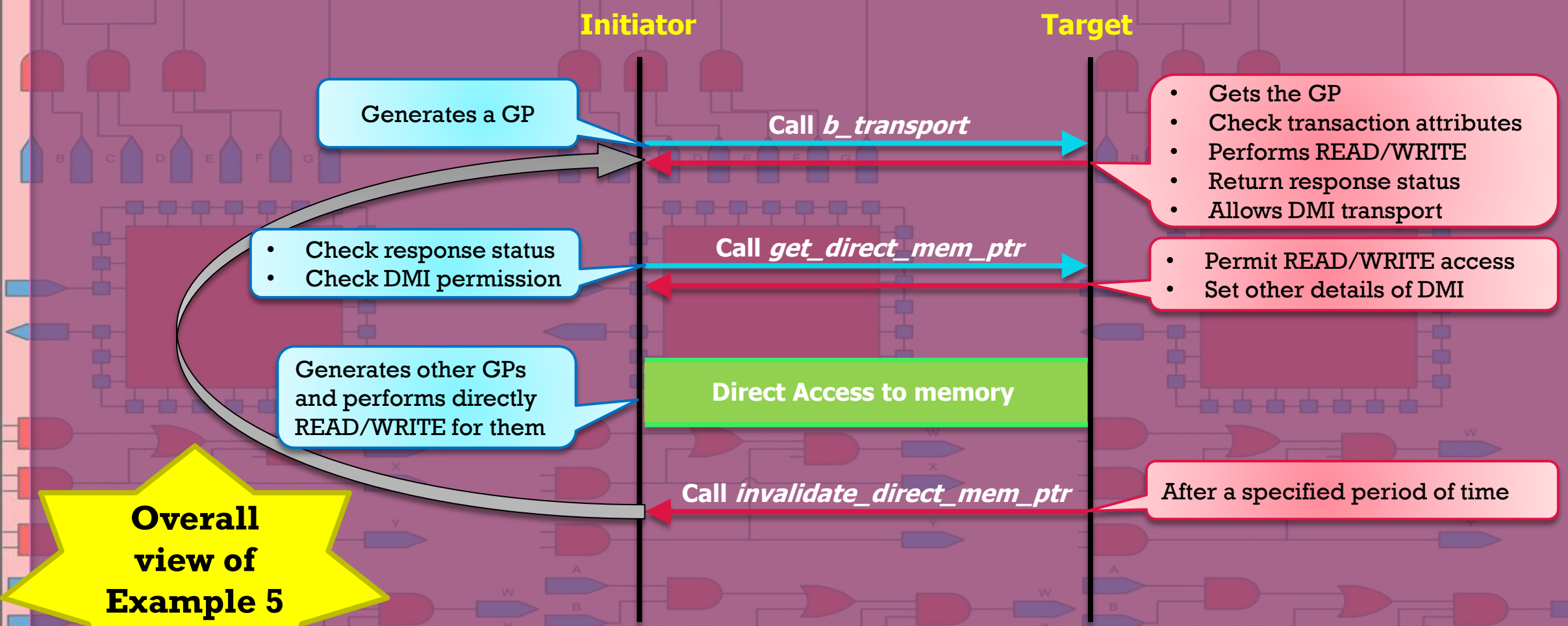
The attributes are used by initiator

```
  private:
    unsigned char*    m_dmi_ptr;
    sc_dt::uint64     m_dmi_start_address;
    sc_dt::uint64     m_dmi_end_address;
    dmi_access_e      m_dmi_access;
    sc_core::sc_time  m_dmi_read_latency;
    sc_core::sc_time  m_dmi_write_latency;
};
}
```

If the forward call is successful, the target returns the *dmi_ptr*, which must point to the data element corresponding to the *dmi_start_address*

The absolute start and end addresses of the DMI region

DMI Transport



DMI Transport

- Example 5 scenario:
  - The target grants both read and write accesses
  - The entire memory contents have been granted as the DMI region
  - The *b_transport* method calls *set_dmi_allowed* to set the DMI hint. Because there is no point in the initiator making repeated calls to *get_direct_mem_ptr* if it can be told in advance that such calls are going to fail
  - The initiator requests a DMI pointer from the target, knowing the request is likely to succeed
  - For implementing *invalidate_direct_mem_ptr*, the initiator ignores the bounds of the direct memory region, and simply invalidates the DMI pointer whatever

DMI Transport

- Example 5: DMImemoryRW: Initiator, *memWriterReader*

memWriterReader.h

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

(Global Scope)

```
1  #include <systemc.h>
2  #include "tlm.h"
3  #include "tlm_utils/simple_initiator_socket.h"
4
5  class memWriterReader : public sc_module {
6  public:
7     tlm_utils::simple_initiator_socket<memWriterReader> memWRSocket;
8
9     SC_CTOR(memWriterReader) : memWRSocket("mem_WR_socket"), DMIptrValid(false)
10    {
11       memWRSocket.register_invalidate_direct_mem_ptr(this, &memWriterReader::invalidate_direct_mem_ptr);
12       WriteRead = new tlm::tlm_generic_payload;
13
14       SC_THREAD(DMImemWR);
15    }
16
17    void DMImemWR();
18    virtual void invalidate_direct_mem_ptr(sc_dt::uint64, sc_dt::uint64);
19
20    tlm::tlm_generic_payload* WriteRead;
21    int data;
22    bool DMIptrValid;
23    tlm::tlm_dmi DMIdata;
24 };
```

Defining the initiator socket and specializing it for *memWriterReader*

defaults to 32-bits wide

Defining a transaction with Generic Payload type

Implement the functionality of the initiator, i.e., generating transactions and sending them

Register callback for incoming *invalidate_direct_mem_ptr* interface method call

Defining a DMI transaction object

DMI Transport

memWriterReader.cpp

- Example 5: DMImemoryRW: Initiator, *memWriterReader*

memWriterReader.h | memWriterReader.cpp | memoryUnit.h

(Global Scope)

```
1  #include "memWriterReader.h"
2
3  void memWriterReader::DMImemWR()
4  {
5      sc_time processTime = sc_time(10, SC_NS);
6      for (int i = 0; i < 128; i += 4) {
7          tlm::tlm_command cmd = (tlm::tlm_command) (rand() % 2);
8          if (cmd == tlm::TLM_WRITE_COMMAND) data = 0xFF000000 | i;
9          if (DMIptrValid) {
10             if (cmd == tlm::TLM_READ_COMMAND) {
11                 assert(DMIdata.is_read_allowed());
12                 memcpy(&data, DMIdata.get_dmi_ptr() + i, 4);
13                 wait(DMIdata.get_read_latency());
14             }
15             else if (cmd == tlm::TLM_WRITE_COMMAND) {
16                 assert(DMIdata.is_write_allowed());
17                 memcpy(DMIdata.get_dmi_ptr() + i, &data, 4);
18                 wait(DMIdata.get_write_latency());
19             }
20
21             cout << "DMI    = { " << (cmd ? 'W' : 'R') << ", " << hex << i
22                  << " } , data = " << hex << data << " at time " << sc_time_stamp() << endl;
23         }
24         else {
25             WriteRead->set_command(cmd);
26             WriteRead->set_address(i);
27             WriteRead->set_data_ptr((unsigned char*)&data);
28             WriteRead->set_data_length(4);
29             WriteRead->set_streaming_width(4);
30             WriteRead->set_byte_enable_ptr(0);
31             WriteRead->set_dmi_allowed(false);
32             WriteRead->set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);
```

Generate a random sequence of reads and writes

Reused across calls to b_transport and DMI

Bypass transport interface and use DMI if it is available

If DMI is not available, set GP parameters

memWriterReader.h | memWriterReader.cpp | memoryUnit.h | memoryUnit.cpp | memWriterRe

Calling the *b_transport* method

```
memWRSocket->b_transport(*WriteRead, processTime);

if (WriteRead->is_response_error()) {
    char txt[100];
    sprintf(txt, "Error from b_transport, response status = %s",
        WriteRead->get_response_string().c_str());
    SC_REPORT_ERROR("TLM-2", txt);
}
```

Initiator obliged to check response status

Check DMI hint

```
if (WriteRead->is_dmi_allowed()) {
    DMIdata.init();
    DMIptrValid = memWRSocket->get_direct_mem_ptr(*WriteRead, DMIdata);
}

cout << "trans = { " << (cmd ? 'W' : 'R') << ", " << hex << i
    << " } , data = " << hex << data << " at time " << sc_time_stamp()
    << " delay = " << processTime << endl;
```

Implement backward DMI method

```
void memWriterReader::invalidate_direct_mem_ptr(sc_dt::uint64 start_range,
                                                sc_dt::uint64 end_range) {
    DMIptrValid = false;
}
```

DMI Transport

- Example 5: DMImemoryRW: Target, *memoryUnit*

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

(Global Scope)

```
1  #include <systemc.h>
2  #include "tlm.h"
3  #include "tlm_utils/simple_target_socket.h"
4
5  class memoryUnit : public sc_module {
6  public:
7     tlm_utils::simple_target_socket<memoryUnit> memSocket;
8
9     static const int SIZE = 256;
10    int memArray[SIZE];
11    const sc_time Latency = sc_time(10, SC_NS);
12
13    SC_CTOR(memoryUnit) : memSocket("memory_side_socket"){
14       memSocket.register_b_transport(this, &memoryUnit::b_transport);
15       memSocket.register_get_direct_mem_ptr(this, &memoryUnit::get_direct_mem_ptr);
          for (int i = 0; i < SIZE; i++)
              memArray[i] = 0xAA000000 | (rand() % 256);
          SC_THREAD(DMImem);
       }
       virtual void b_transport(tlm::tlm_generic_payload&, sc_time&);
23     virtual bool get_direct_mem_ptr(tlm::tlm_generic_payload&, tlm::tlm_dmi&);
24     void DMImem();
25 };
```

memoryUnit.h

Defining the target socket and specializing it for *memoryUnit*

Register callback for incoming *b_transport* interface method call

Register callback for incoming *get_direct_mem_ptr* interface method call

Initializing memory

Calling *invalidate_direct_mem_ptr*

DMI Transport

memoryUnit.cpp

```
1   #include "memoryUnit.h"
2
3
4   void memoryUnit::b_transport(tlm::tlm_generic_payload& gotThis, sc_time& delayValue)
5   {
6     tlm::tlm_command cmd = gotThis.get_command();
7     uint64           adr = gotThis.get_address()/4;
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 > 4 || 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        memcpy(ptr, &memArray[adr], len);
19    }
20    else if (cmd == tlm::TLM_WRITE_COMMAND){
21        memcpy(&memArray[adr], ptr, len);
22    }
23    wait(delayValue);
24    gotThis.set_dmi_allowed(true);
25    gotThis.set_response_status(tlm::TLM_OK_RESPONSE);
26  }
```

- Example 5: DMImemoryRW: Target, *memoryUnit*

Getting GP parameters

Obliged to check address range and check for unsupported features

Obliged to implement read and write commands

Set DMI hint to indicated that DMI is supported

Obliged to set response status to indicate successful completion

DMI Transport

⊙ Example 5: DMImemoryRW: Target, *memoryUnit* (Cont.)

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

(Global Scope)

```
28  bool memoryUnit::get_direct_mem_ptr(tlm::tlm_generic_payload& gotThis, tlm::tlm_dmi& dmi_data)
    {
        dmi_data.allow_read_write();

        dmi_data.set_dmi_ptr((unsigned char*)&memArray[0]);
        dmi_data.set_start_address( 0 );
        dmi_data.set_end_address( SIZE*4-1 );
        dmi_data.set_read_latency(Latency);
        dmi_data.set_write_latency(Latency);

36      return true;
37  }
38
39  void memoryUnit::DMImem()
40  {
        for (int i = 0; i < 4; i++) {
            wait(Latency * 8);
            memSocket->invalidate_direct_mem_ptr(0, SIZE - 1);
        }
47  }
```

Permit read and write access

Set other attributes of the DMI object to describe the details of the access

Invalidate DMI pointers periodically

memoryUnit.cpp

The *dmi_ptr* is the actual direct memory pointer. The target is free to grant any DMI region that encloses the requested address. *start_address* and *end_address* describe the bounds of the DMI region from the point of view of the target

DMI Transport

- Example 5: DMImemoryRW: 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

Bind initiator socket to target socket

DMI Transport

- Example 5: DMImemoryRW: Main

memWriterReader_Main.cpp | memWriterReader_TB.cpp | memWriterReader.h

(Global Scope)

```
1   #include "memWriterReader_TB.h"
2
3   int sc_main(int argc, char* argv[])
4   {
5       memWriterReader_TB TB1("memWriterReader_TB");
6       sc_start();
7       return 0;
8   }
```

memWriterReader_Main.cpp

DMI Transport

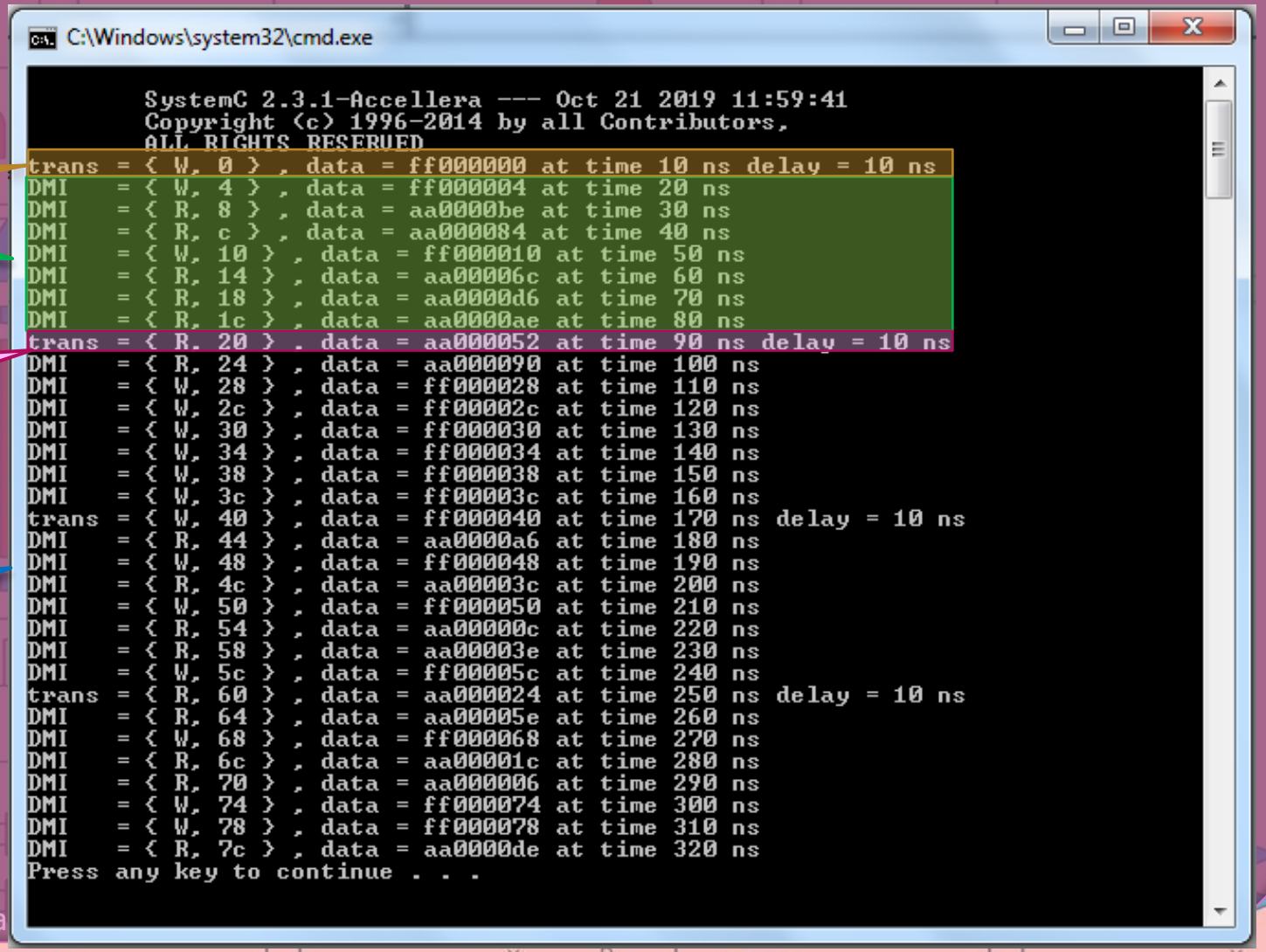
- Example 5:
- DMImemoryRW: Output

The transaction is done by *b_transport*

After the completion of a transaction, DMI access is provided for 80 NS

DMI access is invalidated after 80 NS. So, The transaction is done by *b_transport*

32 (128/4) transactions are generated and sent

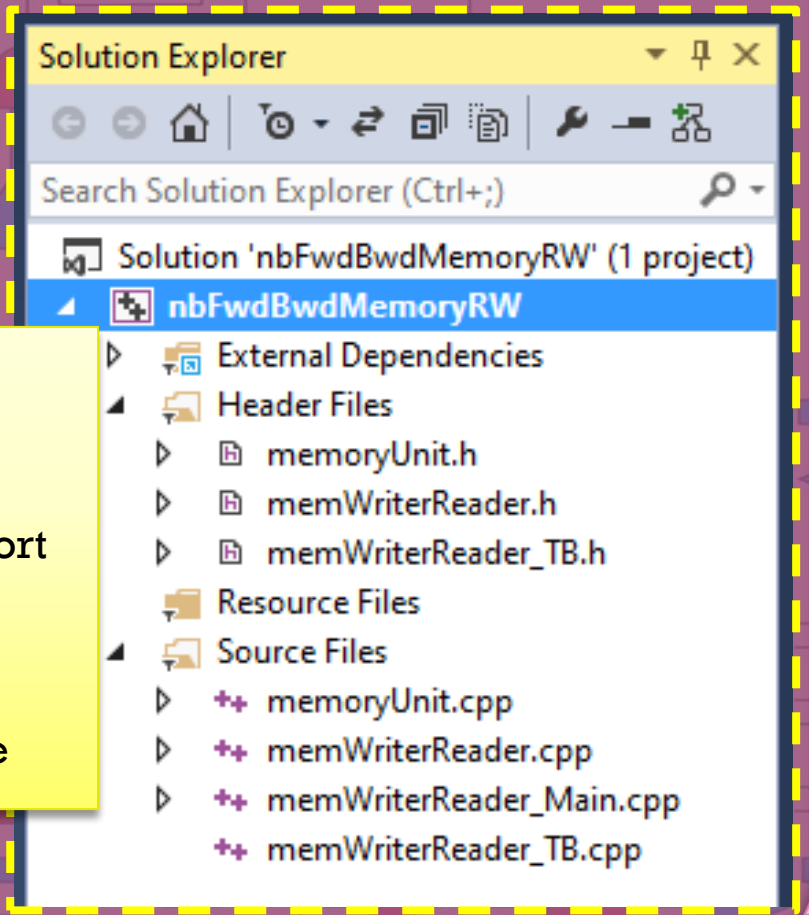


Transaction Level Modeling

- 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
      - nbFwdBwdMemoryRW Example
- + Complete System

The concepts covered:

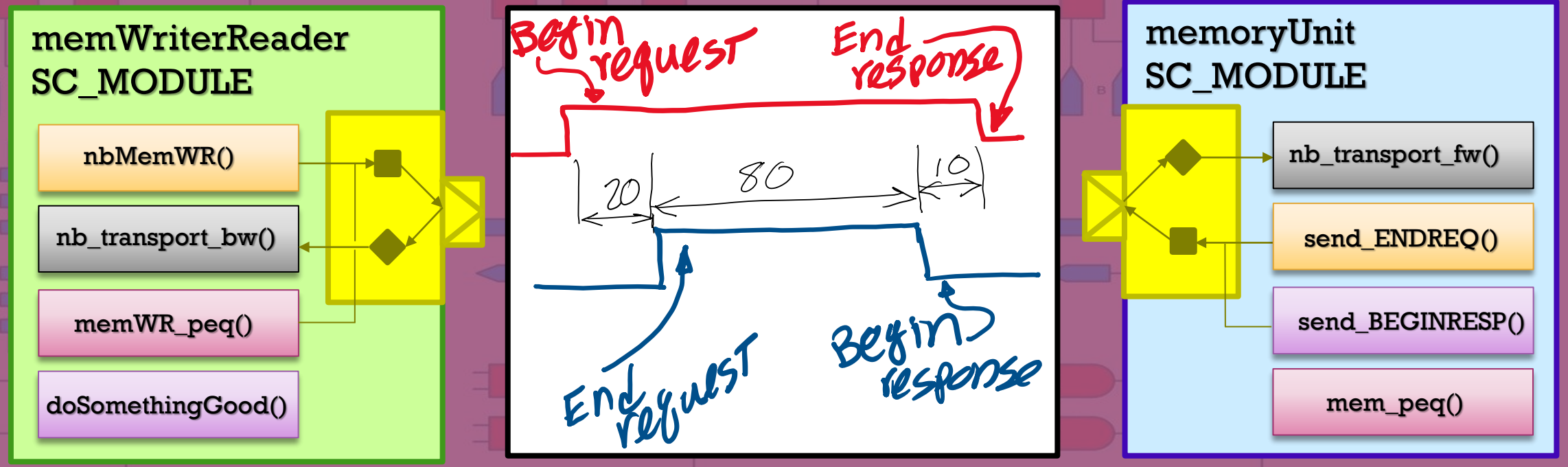
- ✓ Generic payload
- ✓ Simple sockets
- ✓ Non-blocking transport
  - ✓ Forward path
  - ✓ Backward path
  - ✓ 4 calls
- ✓ Payload Event Queue



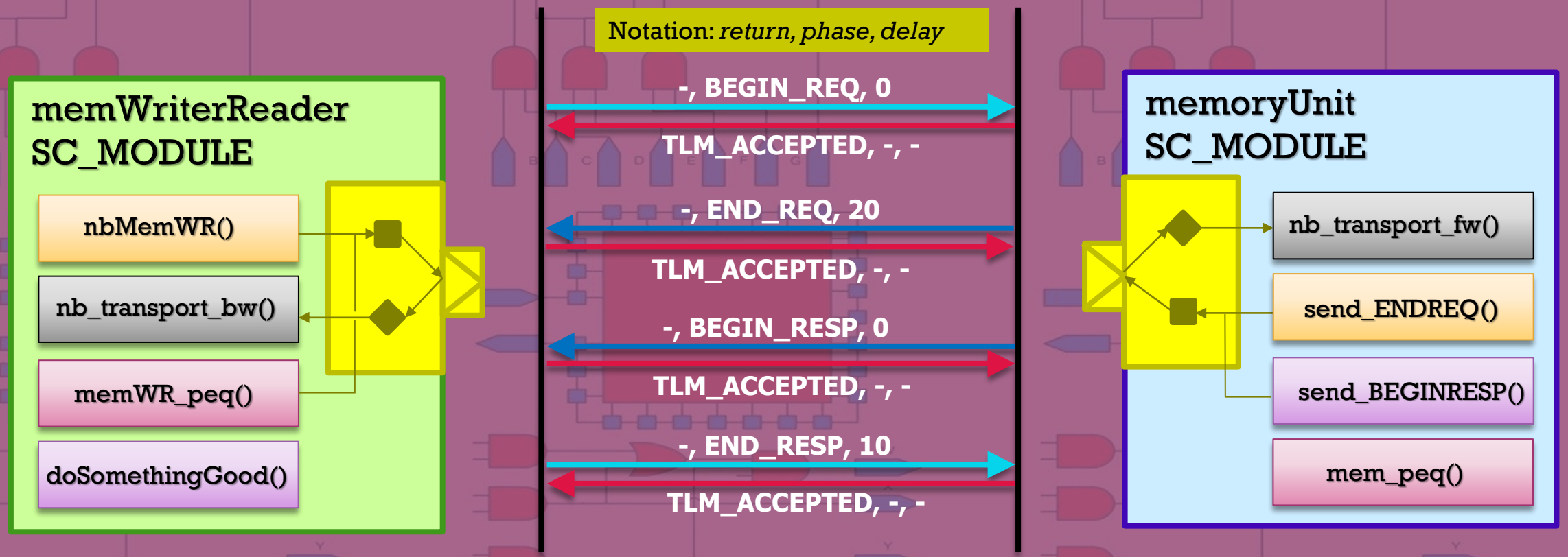
Payload Event Queue (PEQ)

- Is a class that maintains a queue of SystemC event notifications, where each notification carries an associated transaction object
- Each transaction is written into the PEQ annotated with a delay, and each transaction emerges from the back of the PEQ at a time calculated from the current simulation time plus the annotated delay
- Will schedule the timing point associated with the nb_transport call to occur at the correct simulation time
- There are two ways to implement PEQ
  - peq_with_get
  - peq_with_cb_and_phase

Non-blocking Transport: Forward & Backward

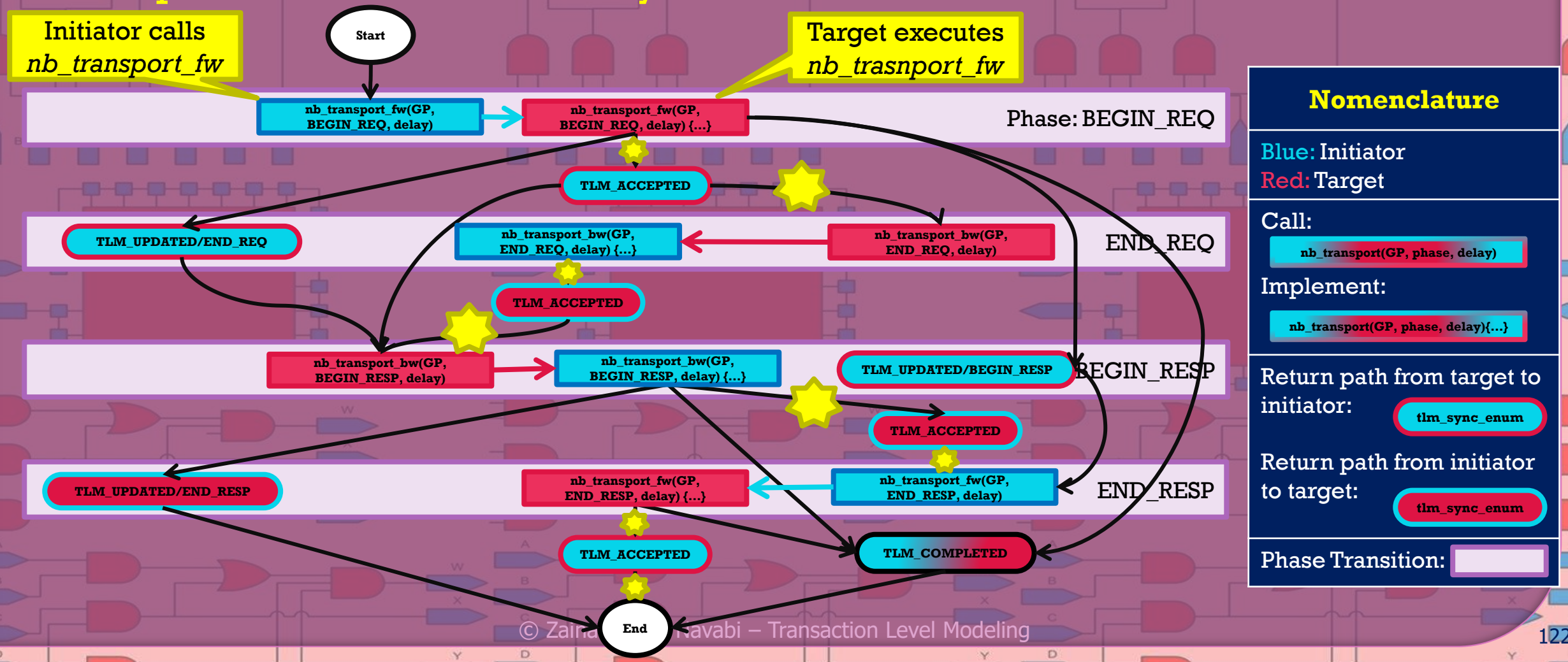


Non-blocking Transport: Forward & Backward



Non-blocking Transport: Forward & Backward

- Example 6 – nbFwdBwdMemoryRW: Base Protocol Flow Control



Non-blocking Transport: Forward & Backward

- Example 6 – nbFwdBwdMemoryRW: 4 calls

Non-blocking Transport: Forward & Backward

⊙ Example 6 – nbFwdBwdMemoryRW: Initiator, *memWriterReader*

memWriterReader.cpp | memWriterReader.h | memoryUnit.cpp | memory

(Global Scope)

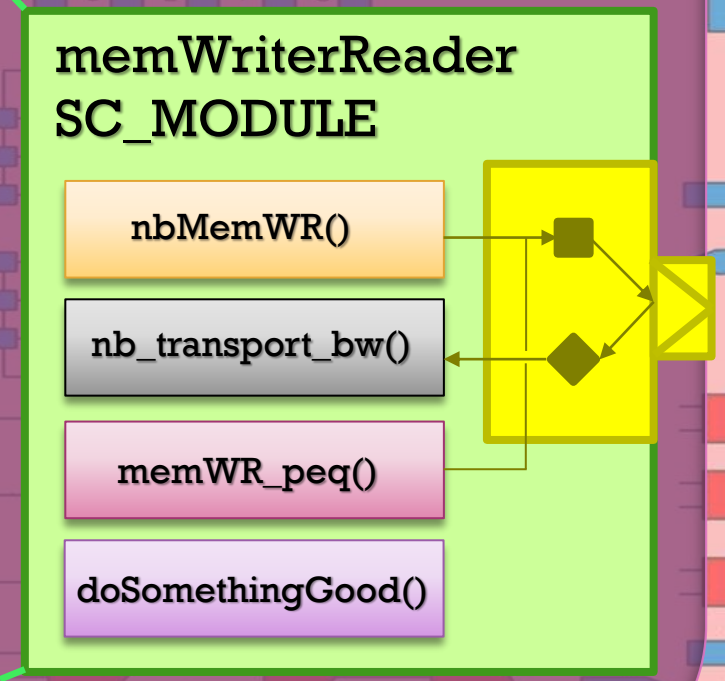
```
1  #include <systemc.h>
2  #include "tlm.h"
3  #include "tlm_utils/simple_initiator_socket.h"
4  #include "tlm_utils/peq_with_cb_and_phase.h"
5
6  class memWriterReader : public sc_module {
7  public:
8      tlm_utils::simple_initiator_socket<memWriterReader, 32> memWRSocket;
9
10     SC_CTOR(memWriterReader) : memWRSocket("mem_WR_socket"), nBlockWriteRead(0),
11                                m_peq(this, &memWriterReader::memWR_peq){
12         nBlockWriteRead = new tlm::tlm_generic_payload();
13         SC_THREAD(nbMemWR);
14         memWRSocket.register_nb_transport_bw(this, &memWriterReader::nb_transport_bw);
15     }
16     tlm::tlm_generic_payload* nBlockWriteRead;
17     void nbMemWR();
18     void doSomethingGood(tlm::tlm_generic_payload& );
19     tlm_utils::peq_with_cb_and_phase<memWriterReader> m_peq;
20     void memWR_peq(tlm::tlm_generic_payload&, const tlm::tlm_phase& );
21     virtual tlm::tlm_sync_enum nb_transport_bw(tlm::tlm_generic_payload&, tlm::tlm_phase&, sc_time&);
22
23     int data[5];
24 };
```

memWriterReader.h

Register callback for incoming *nb_transport_bw* interface method call

Non-blocking transport, so the initiator can process the data of the incoming transaction

Declared a Payload Event Queue, then implement it



Non-blocking Transport: Forward & Backward

memWriterReader.cpp | memWriterReader.h | memoryUnit.cpp | memory

memWriterReader

```cpp
#include "memWriterReader.h"

void memWriterReader::nbMemWR(){
    tlm::tlm_phase forwardPhase;
    sc_time delay;
    for (int i = 0; i < 25; i = i + 5){
        tlm::tlm_command cmd = (tlm::tlm_command)(rand() % 2);
        if (cmd == tlm::TLM_WRITE_COMMAND) {
            data[0] = (i+1); data[1] = (i+2); data[2] = (i+3);
            data[3] = (i+4); data[4] = (i+5);
        }
        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;
        delay = sc_time(0, SC_NS);
        cout << "************************************************************************************ " << endl;
        cout << "==> The initator sends BEGIN_REQ for " << (cmd ? 'W' : 'R');
        if (cmd == 1) {cout << " data: ";
        for (int j = 0; j<5; j++) cout << data[j] << " "; }
        cout << " at " << sc_time_stamp() <<" delay=" << delay << '\n';

        tlm::tlm_sync_enum returnStatus;
        returnStatus = memWRSocket->
            nb_transport_fw(*nBlockWriteRead, forwardPhase, delay);
        cout << "<-- Return path for BEGIN_REQ is " << returnStatus << " at " << sc_time_stamp() << endl;
        wait(130 , SC_NS);
    }
}
```

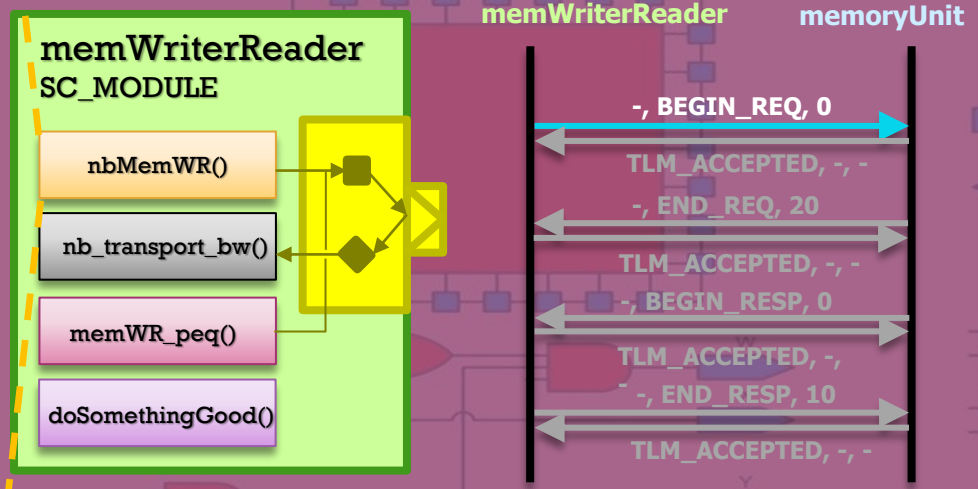
memWriterReader.cpp

Generates a generic payload

Call *nb_transport_fw* for *BEGIN_REQ*

Do some operations
Wait enough for avoiding pipeline transactions

- Example 6 - nbFwdBwdMemoryRW: Initiator, *memWriterReader*



Non-blocking Transport: Forward & Backward

- Example 6 – nbFwdBwdMemoryRW: Initiator, *memWriterReader*

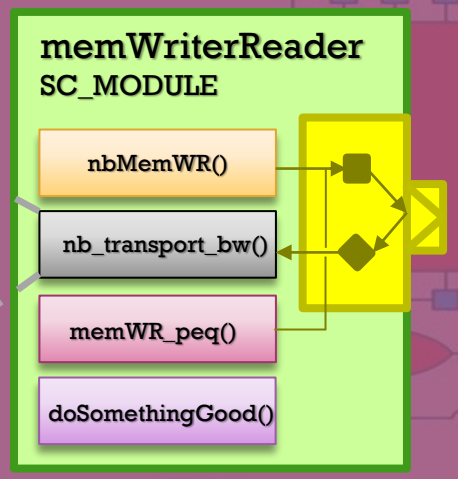
memWriterReader.cpp

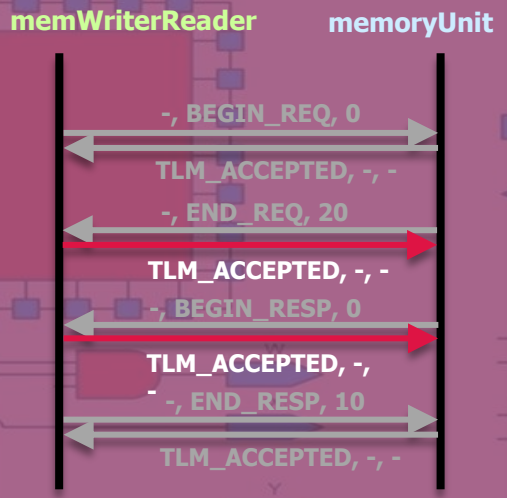
memWriterReader.cpp | memWriterReader.h | memoryUnit.cpp | memoryUnit.h

memWriterReader | nbMemWR()

```
68  tlm::tlm_sync_enum memWriterReader::nb_transport_bw(tlm::tlm_generic_payload& trans,
69      tlm::tlm_phase& phase, sc_time& delay){
70    m_peq.notify(trans, phase, delay);
71    return tlm::TLM_ACCEPTED;
72  }
```

Move the incoming transactions to PEQ, then return *TLM_ACCEPTED*





Non-blocking Transport: Forward & Backward

⊙ Example 6 – nbFwdBwdMemoryRW: Initiator, *memWriterReader*

Payload event queue callback to handle transactions from target
Transaction could have arrived through return path or backward path

memWriterReader.cpp

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

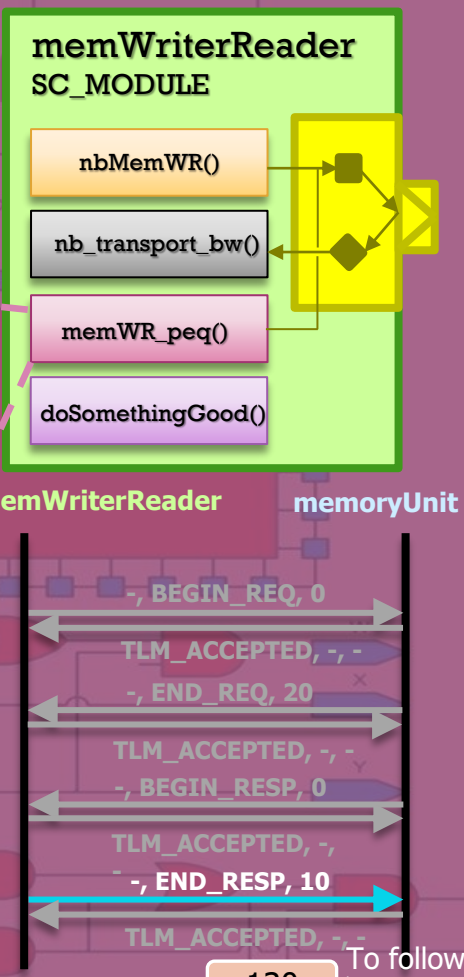
memWriterReader | nbMemWR()

```
52  void memWriterReader::memWR_peq(tlm::tlm_generic_payload& trans, const tlm::tlm_phase& phase){
53
54      if (phase == tlm::BEGIN_REQ || phase == tlm::END_RESP)
55          SC_REPORT_FATAL("TLM-2", "Illegal transaction phase received by initiator");
56      else if (phase == tlm::BEGIN_RESP){
57          tlm::tlm_phase forwardPhase = tlm::END_RESP;
58          sc_time delay = sc_time(10, SC_NS);
59          cout << "==> The initator sends END_RESP at " << sc_time_stamp() << " delay=" << delay << endl;
60          tlm::tlm_sync_enum returnStatus =
61              memWRSocket->nb_transport_fw(trans, forwardPhase, delay);
62          cout << "<-- Return path for END_RESP is " << returnStatus << " at " << sc_time_stamp() << endl;
63          doSomethingGood(*nBlockWriteRead);
64      }
65  }
```

Response accept delay = 10ns

Call *nb_transport_fw* for *END_RES*

The initiator can process the data of the incoming transaction



Non-blocking Transport: Forward & Backward

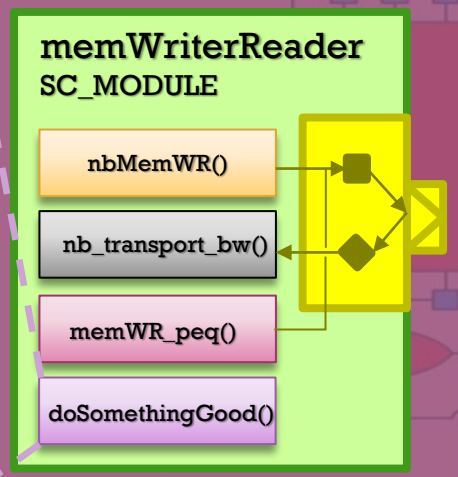
- Example 6 – nbFwdBwdMemoryRW: Initiator, *memWriterReader*

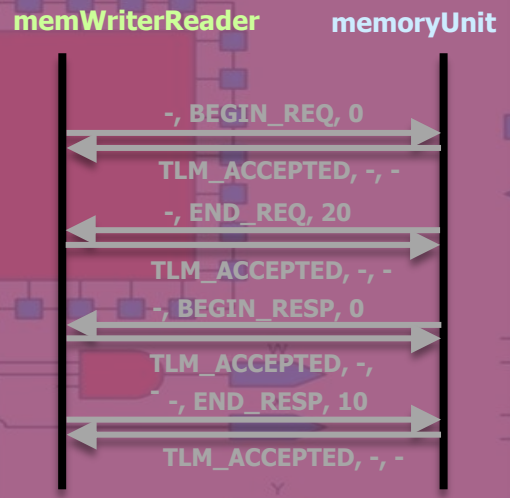
memWriterReader.cpp

```
memWriterReader.cpp | memWriterReader.h | memoryU
memWriterReader                          memWR_peq(tlm:
37  void memWriterReader::doSomethingGood( tlm::tlm_generic_payload& completeTrans)
38  {
    if ( completeTrans.is_response_error() )
        SC_REPORT_ERROR("TLM-2", "error...\n");
    tlm::tlm_command cmd = completeTrans.get_command();
    uint64           adr = completeTrans.get_address();
    int*             ptr = reinterpret_cast<int*>( completeTrans.get_data_ptr() );
    if (cmd == tlm::TLM_READ_COMMAND){
        cout << "The incoming data from memArray is ready to process: ";
        for (int j = 0; j < 5; j++) cout << *(ptr + j) << " ";
        cout << "at " << sc_time_stamp() << '\n';
    }
46
47
48
49  }
```

Initiator obliged to check response status on receiving *BEGIN_RESP* or *TLM_COMPLETED*

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





Non-blocking Transport: Forward & Backward

⊙ Example 6 – nbFwdBwdMemoryRW: Target, *memoryUnit*

memoryUnit.cpp | memoryUnit.h

(Global Scope)

```
1  #include <systemc.h>
2  #include "tlm.h"
3  #include "tlm_utils/simple_target_socket.h"
4  #include "tlm_utils/peq_with_cb_and_phase.h"
5
6  class memoryUnit : public sc_module {
7  public:
8      tlm_utils::simple_target_socket<memoryUnit, 32> memSocket;
9      static const int SIZE=32;
10
11     SC_CTOR(memoryUnit) : memSocket("memory_side_socket"),
12         m_peq(this, &memoryUnit::mem_peq) {
13
14         memSocket.register_nb_transport_fw(this, &memoryUnit::nb_transport_fw);
15         for (int i = 0; i < SIZE; i++)
16             memArray[i] = (i + 82);
17     }
18     virtual tlm::tlm_sync_enum nb_transport_fw( tlm::tlm_generic_payload&, tlm::tlm_phase&, sc_time& );
19     tlm_utils::peq_with_cb_and_phase<memoryUnit> m_peq;
20     void mem_peq(tlm::tlm_generic_payload&, const tlm::tlm_phase& );
21     void send_ENDREQ(tlm::tlm_generic_payload&);
22     void send_BEGINRESP(tlm::tlm_generic_payload& );
23     int memArray[SIZE];
24 };
```

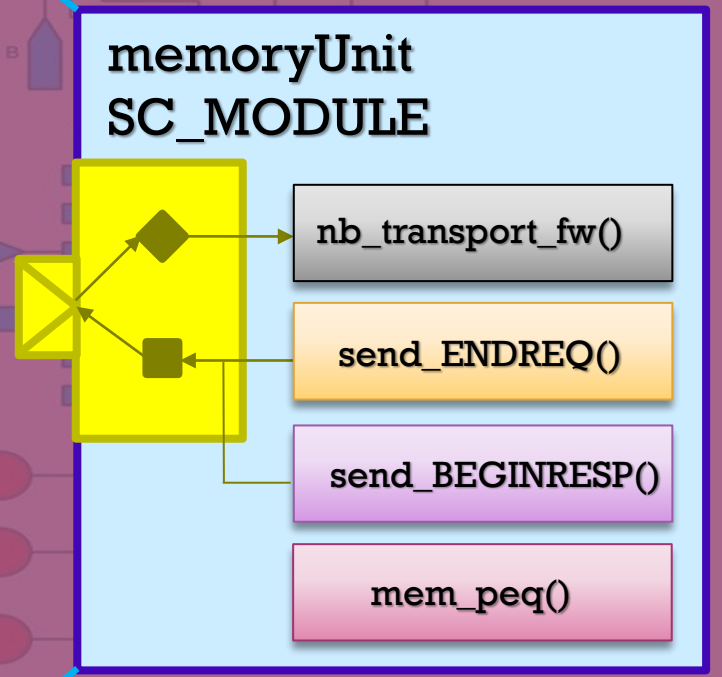
memoryUnit.h

Register callback for incoming *nb_transport_fw* interface method call

Declared a Payload Event Queue, then implement it

Function to call *nb_transport_bw* for *END_REQ*

Function to call *nb_transport_bw* for *END_REQ*



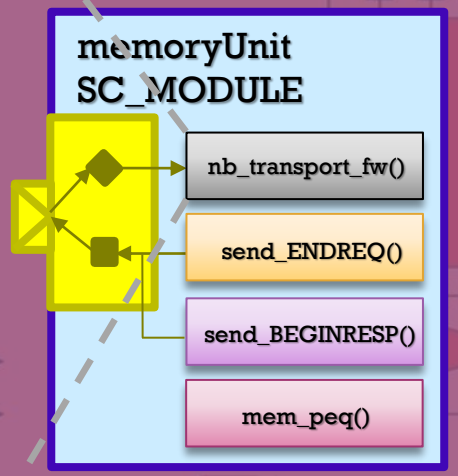
Non-blocking Transport: Forward & Backward

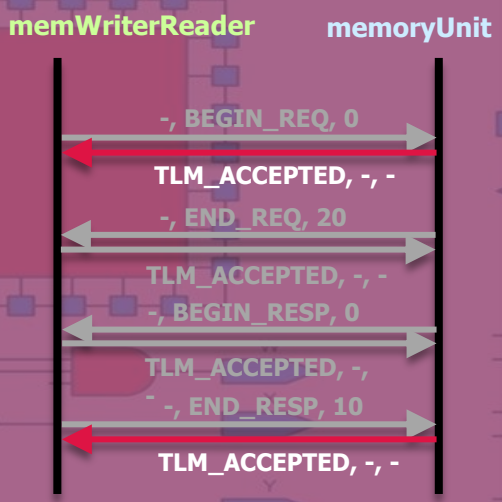
- Example 6 – nbFwdBwdMemoryRW: Target, *memoryUnit*

memoryUnit.cpp

```
memoryUnit.cpp | memoryUnit.h
memoryUnit                                                    BEGINRESP
 1  #include "memoryUnit.h"
 2
 3  tlm::tlm_sync_enum memoryUnit::nb_transport_fw
 4        (tlm::tlm_generic_payload& receivedTrans,
 5         tlm::tlm_phase& phase, sc_time& delay ){
 6
 7     tlm::tlm_command cmd = receivedTrans.get_command();
 8     uint64           adr = receivedTrans.get_address();
 9     unsigned char*   ptr = receivedTrans.get_data_ptr();
10     unsigned int     len = receivedTrans.get_data_length();
11     unsigned char*   byt = receivedTrans.get_byte_enable_ptr();
12     unsigned int     wid = receivedTrans.get_streaming_width();
13     if (byt != 0) {
14         receivedTrans.set_response_status( tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE );
15         return tlm::TLM_COMPLETED;}
16     if (len > 5 || wid < len) {
17         receivedTrans.set_response_status( tlm::TLM_BURST_ERROR_RESPONSE );
18         return tlm::TLM_COMPLETED;}
19
20     m_peq.notify(receivedTrans, phase, delay);
21     return tlm::TLM_ACCEPTED;
22  }
```

Move the incoming transactions to PEQ, then return *TLM_ACCEPTED*





Non-blocking Transport: Forward & Backward

- Example 6 – nbFwdBwdMemoryRW: Target, *memoryUnit*

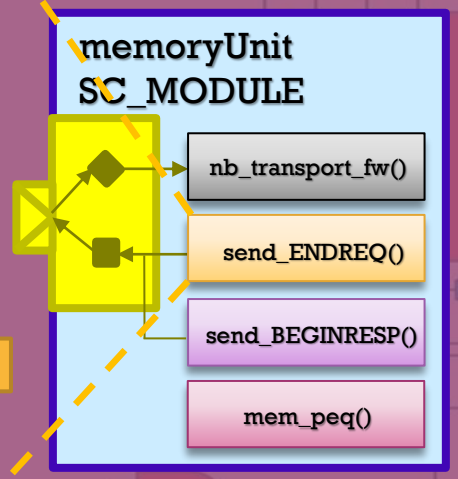
memoryUnit.cpp

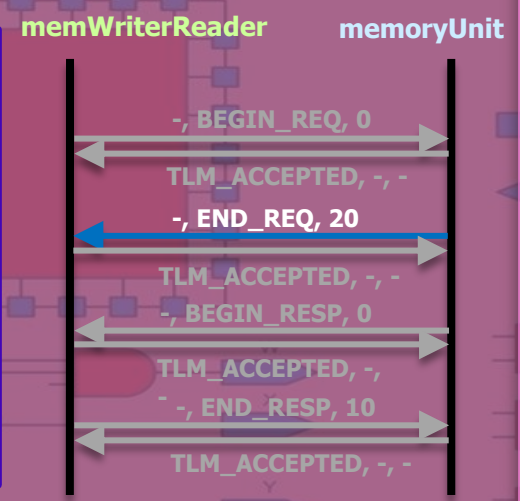
```
memoryUnit.cpp | memoryUnit.h
memoryUnit
46  void memoryUnit::send_ENDREQ(tlm::tlm_generic_payload& receivedTrans){
47      tlm::tlm_sync_enum status;
48      tlm::tlm_phase backwardPhase;
49      sc_time delay;
50
51      backwardPhase = tlm::END_REQ;
52      delay = sc_time(20, SC_NS);
53      cout << "<== The target sends END_REQ at " << sc_time_stamp() << " delay=" << delay << endl;
54      status = memSocket->nb_transport_bw(receivedTrans, backwardPhase, delay);
55      cout << "--> Return path for END_REQ is " << status << " at " << sc_time_stamp() << endl;
56
57      tlm::tlm_phase int_phase = tlm::BEGIN_RESP;
58      delay = delay + sc_time(80, SC_NS);
59      m_peq.notify(receivedTrans, int_phase, delay);
60  }
```

Request accept delay = 20ns

Call *nb_transport_bw* for *END_REQ*

Latency of target = 80+20 =100ns





Non-blocking Transport: Forward & Backward

⊙ Example 6 – nbFwdBwdMemoryRW: Target, *memoryUnit*

memoryUnit.cpp | memoryUnit.h

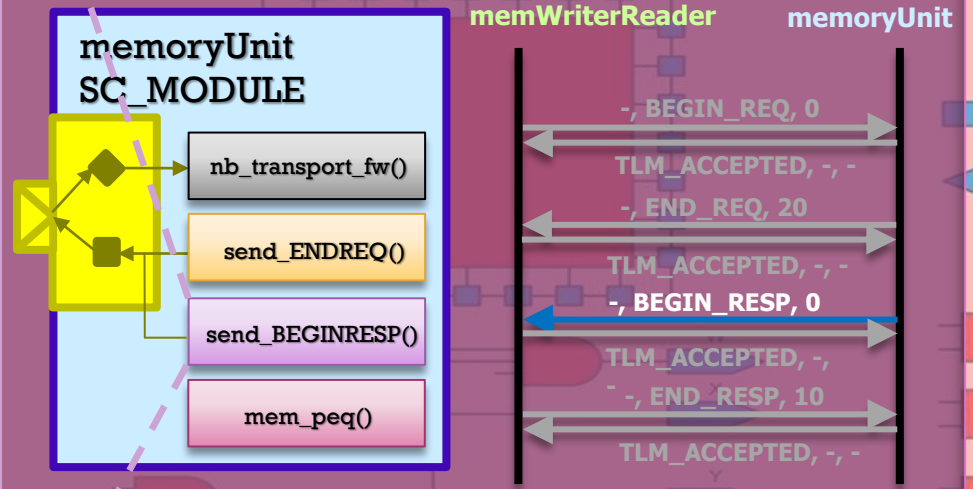
memoryUnit

memoryUnit.cpp

```
63  void memoryUnit::send_BEGINRESP(tlm::tlm_generic_payload& receivedTrans){
64      tlm::tlm_sync_enum status;
65      tlm::tlm_phase backwardPhase;
66      sc_time delay;
67
68      tlm::tlm_command cmd = receivedTrans.get_command();
69      uint64           adr = receivedTrans.get_address();
70      unsigned char*   ptr = receivedTrans.get_data_ptr();
71      unsigned int     len = receivedTrans.get_data_length();
72
73      unsigned int i;
74      if (cmd == tlm::TLM_READ_COMMAND)
75          for (i = 0; i<len; i = i + 1)
76              *(ptr + 4*i) = *((unsigned char*)(memArray + adr + i));
77      else if (cmd == tlm::TLM_WRITE_COMMAND)
78          for (i = 0; i < len; i = i + 1)
79              *((unsigned char*)(memArray + adr + i)) = *(ptr + 4*i);
80
81      receivedTrans.set_response_status(tlm::TLM_OK_RESPONSE);
82      backwardPhase = tlm::BEGIN_RESP;
83      delay = SC_ZERO_TIME;
84      cout << "<== The target sends BEGIN_RESP at " << sc_time_stamp() << " delay=" << delay << endl;
85      status = memSocket->nb_transport_bw(receivedTrans, backwardPhase, delay);
86      cout << "--> Return path for BEGIN_RESP is " << status << " at " << sc_time_stamp() << endl;
87      if (cmd == tlm::TLM_READ_COMMAND) cout << "The data is read from memArray: ";
88      else cout << "The data is written in memArray: ";
89      for (int j = 0; j<5; j++) cout << static_cast<int>(*((unsigned char*)(memArray + adr + j))) << " ";
90      cout << "at " << sc_time_stamp() << endl;
91  }
```

Implement read and write commands

Call *nb_transport_bw* for *BEGIN_RESP*

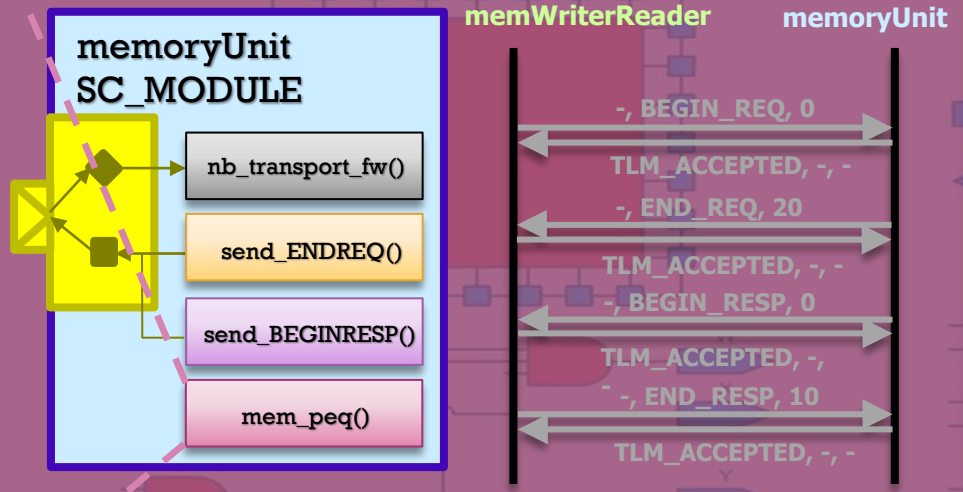


Non-blocking Transport: Forward & Backward

- Example 6 – nbFwdBwdMemoryRW: Target, *memoryUnit*

memoryUnit.cpp

```
memoryUnit.cpp   memoryUnit.h
memoryUnit
25  void memoryUnit::mem_peq(tlm::tlm_generic_payload& receivedTrans, const tlm::tlm_phase& phase){
26      switch (phase) {
27      case tlm::BEGIN_REQ:
28          send_ENDREQ(receivedTrans);
29          break;
30      case tlm::BEGIN_RESP:
31          send_BEGINRESP(receivedTrans);
32          break;
33      case tlm::END_REQ:
34          SC_REPORT_FATAL("TLM-2", "Illegal transaction phase received by target");
35          break;
36      case tlm::END_RESP:
37          cout << "The current transaction was completed at " << sc_time_stamp() << endl;
38          break;
39      default:
40          SC_REPORT_FATAL("TLM-2", "Illegal phase!");
41          break;
42      }
43  }
```



Non-blocking Transport: Forward & Backward

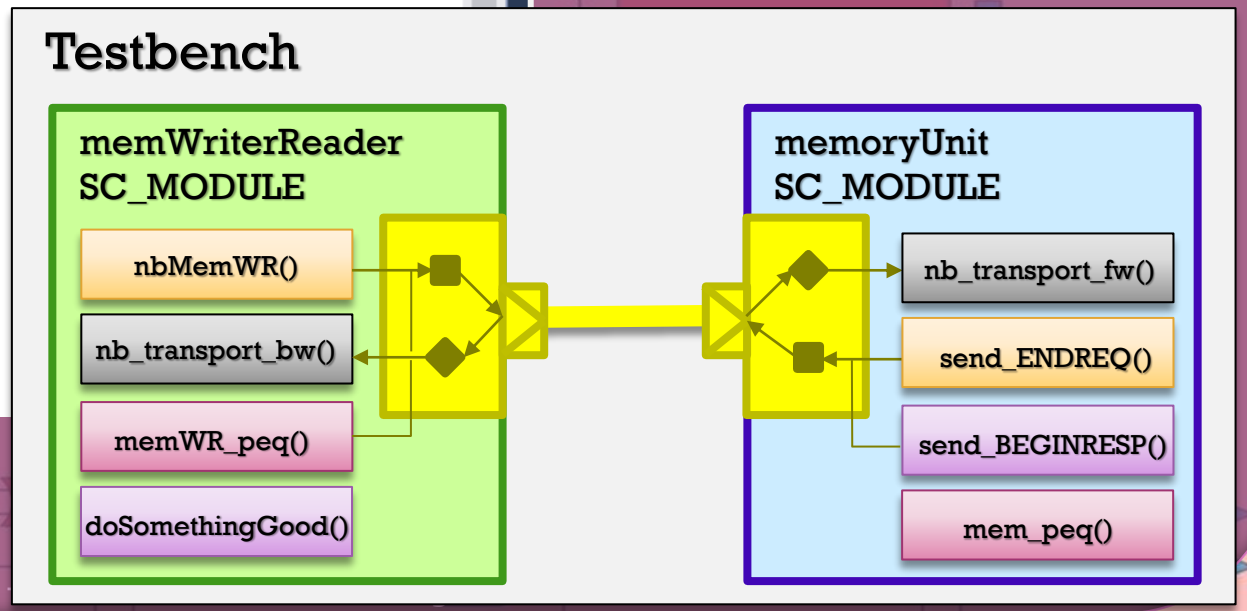
- Example 6 – nbFwdBwdMemoryRW: Testbench

memWriterReader.cpp | memWriterReader_TB.h | memWriterReader.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  };
```

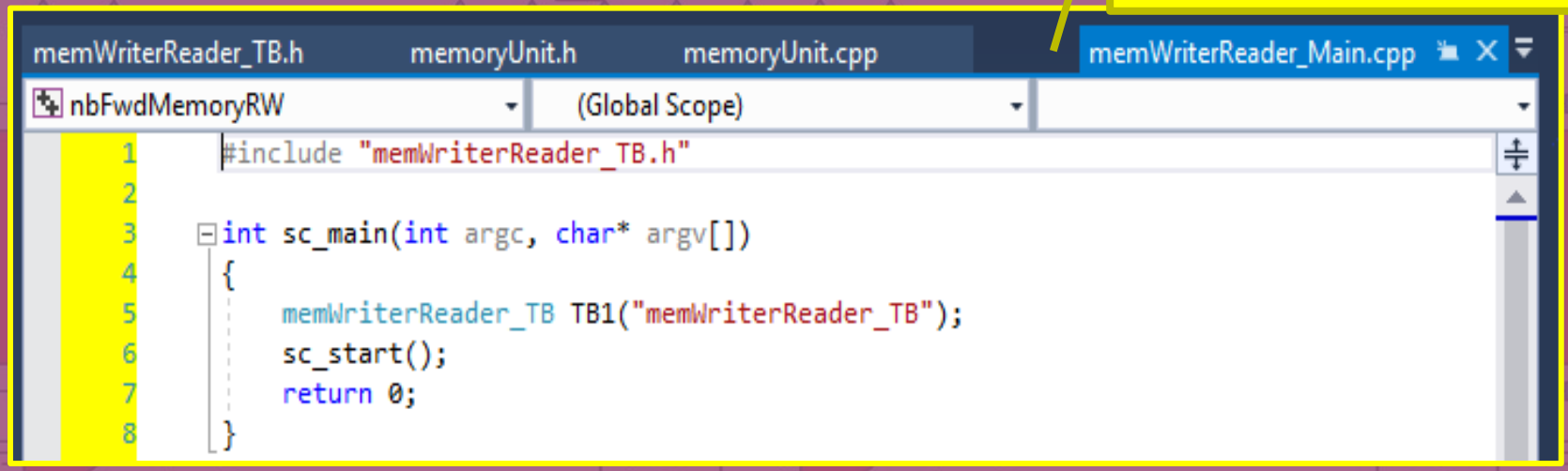
memWriterReader_TB.h



Non-blocking Transport: Forward & Backward

- Example 6 – nbFwdBwdMemoryRW: Main

memWriterReader_Main.cpp



Non-blocking Transport: Forward & Backward

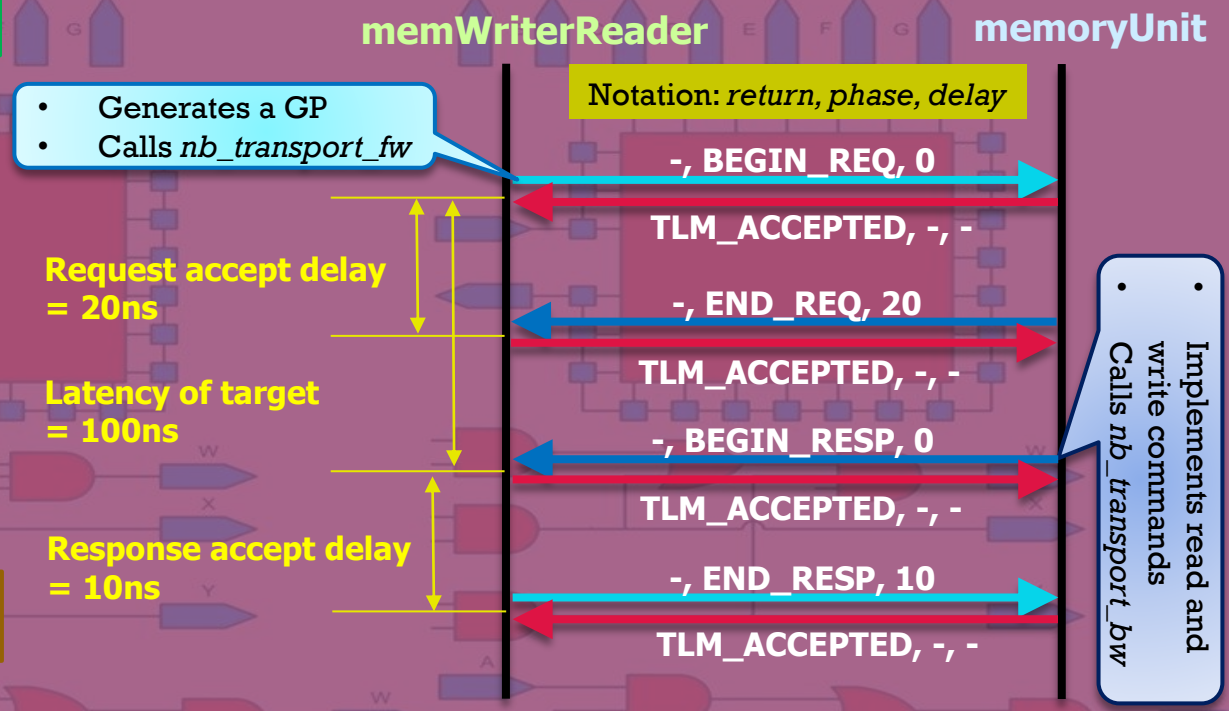
C:\WINDOWS\system32\cmd.exe

```
        SystemC 2.3.1-Accellera --- Sep 29 2019 11:41:37
        Copyright (c) 1996-2014 by all Contributors,
        ALL RIGHTS RESERVED
==> The initiator sends BEGIN_REQ for W data: 1 2 3 4 5  at 0 s delay=0 s
<-- Return path for BEGIN_REQ is 0 at 0 s
<== The target sends END_REQ at 0 s delay=20 ns
--> Return path for END_REQ is 0 at 0 s
<== The target sends BEGIN_RESP at 100 ns delay=0 s
--> Return path for BEGIN_RESP is 0 at 100 ns
The data is written in memArray: 1 2 3 4 5 at 100 ns
==> The initiator sends END_RESP at 100 ns delay=10 ns
<-- Return path for END_RESP is 0 at 100 ns
The current transaction was completed at 110 ns
==> The initiator sends BEGIN_REQ for W data: 6 7 8 9 10  at 130 ns delay=0 s
<-- Return path for BEGIN_REQ is 0 at 130 ns
<== The target sends END_REQ at 130 ns delay=20 ns
--> Return path for END_REQ is 0 at 130 ns
<== The target sends BEGIN_RESP at 230 ns delay=0 s
--> Return path for BEGIN_RESP is 0 at 230 ns
The data is written in memArray: 6 7 8 9 10 at 230 ns
==> The initiator sends END_RESP at 230 ns delay=10 ns
<-- Return path for END_RESP is 0 at 230 ns
The current transaction was completed at 240 ns
==> The initiator sends BEGIN_REQ for R at 260 ns delay=0 s
<-- Return path for BEGIN_REQ is 0 at 260 ns
<== The target sends END_REQ at 260 ns delay=20 ns
--> Return path for END_REQ is 0 at 260 ns
<== The target sends BEGIN_RESP at 360 ns delay=0 s
--> Return path for BEGIN_RESP is 0 at 360 ns
The data is read from memArray: 92 93 94 95 96 at 360 ns
==> The initiator sends END_RESP at 360 ns delay=10 ns
<-- Return path for END_RESP is 0 at 360 ns
The incoming data from memArray is ready to process: 92 93 94 95 96 at 360 ns
The current transaction was completed at 370 ns
```

Writing in the memory

Reading from the memory

- Example 6 – nbFwdBwdMemoryRW: Result

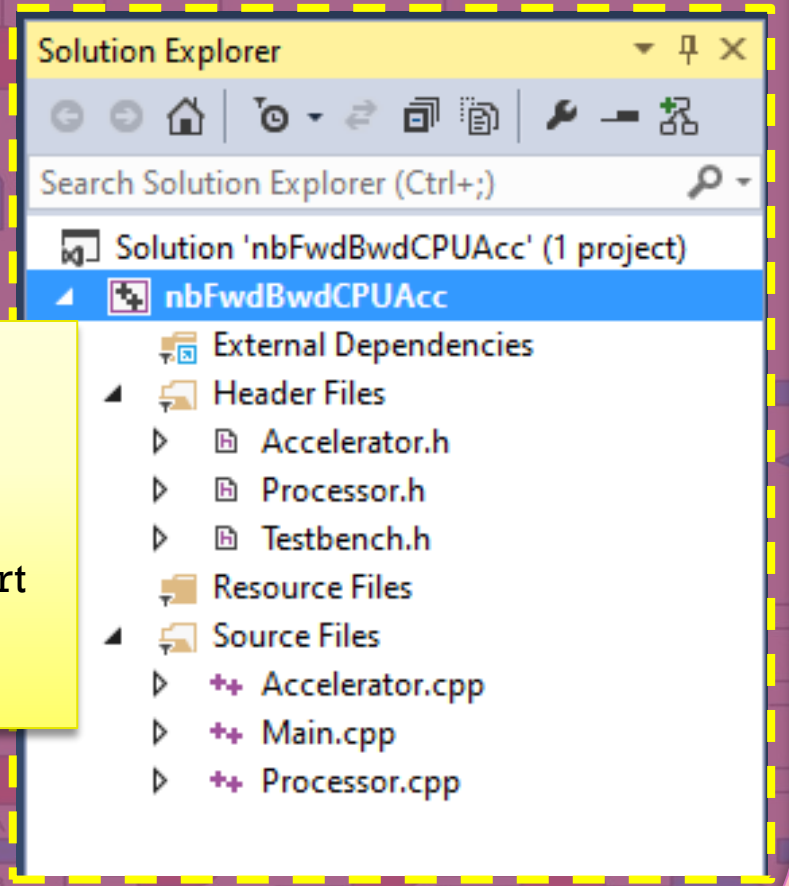


Transaction Level Modeling

- 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
      - nbFwdBwdCPUAcc Example
- + Complete System

The concepts covered:

- ✓ Generic payload
- ✓ Simple sockets
- ✓ Non-blocking transport
  - ✓ Forward path
  - ✓ Backward path

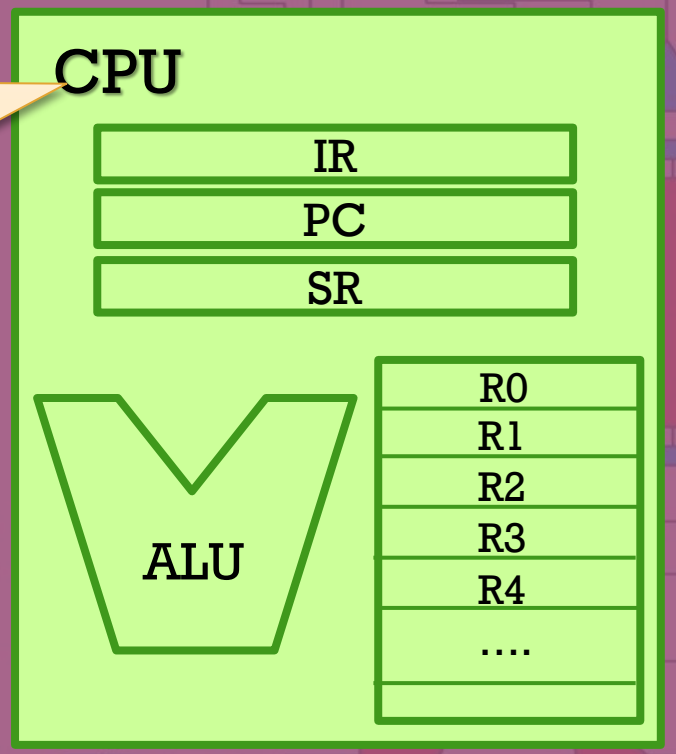


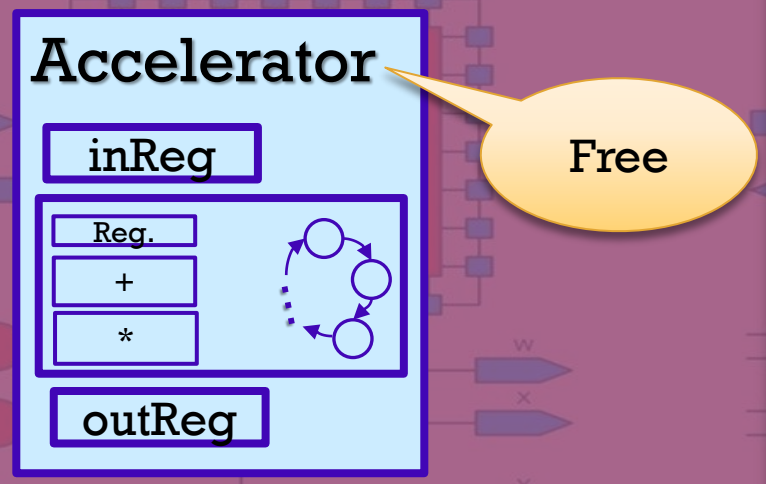
Non-blocking Transport: Forward & Backward

- Example 7 – nbFwdBwdCPUAcc: Scenario

1

Executing instructions Intensive computations



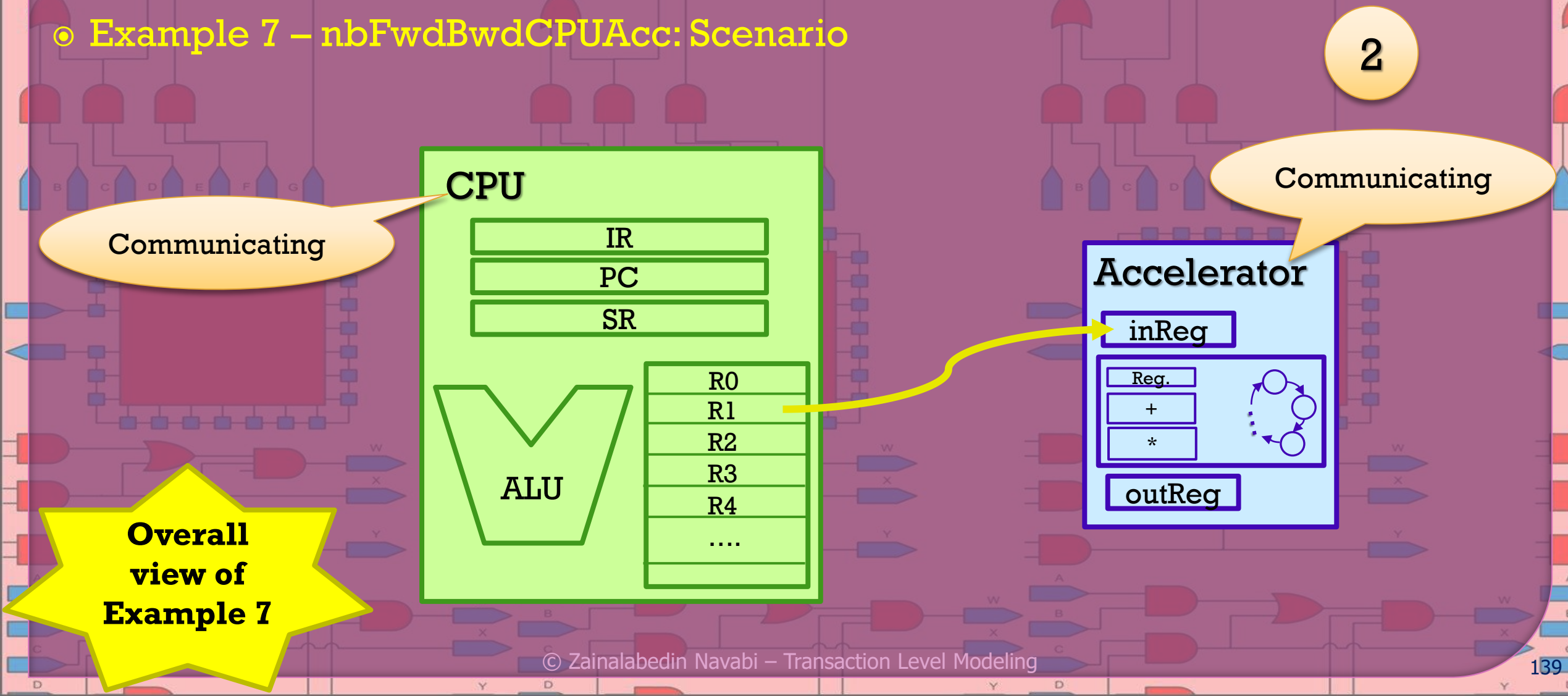


Free

Overall view of Example 7

Non-blocking Transport: Forward & Backward

- Example 7 – nbFwdBwdCPUAcc: Scenario

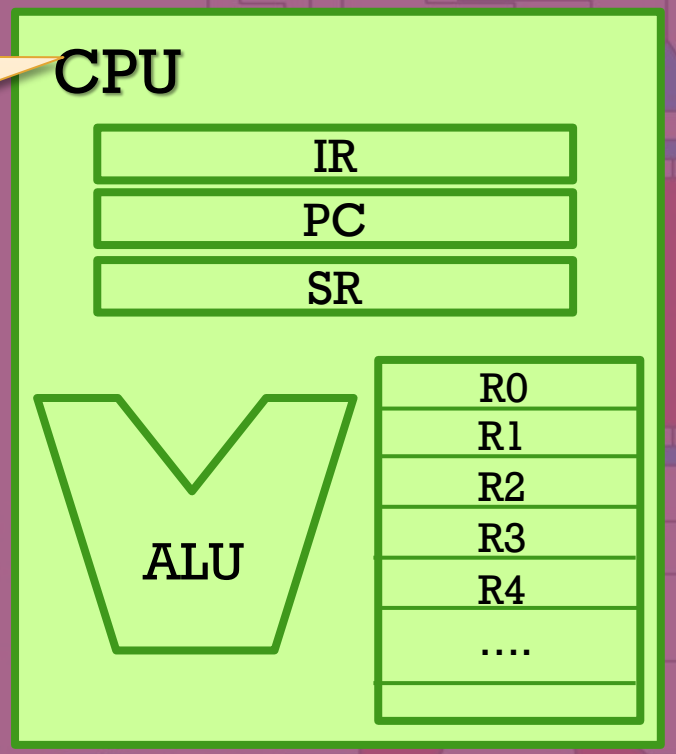


Non-blocking Transport: Forward & Backward

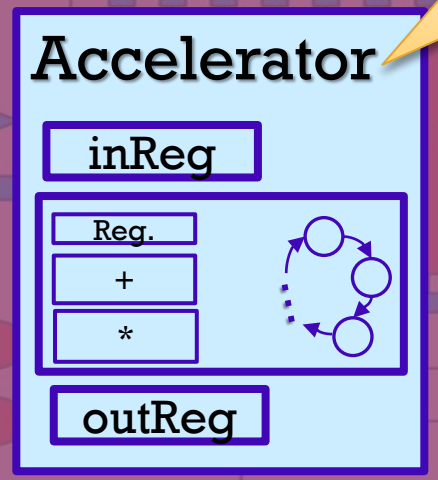
- Example 7 – nbFwdBwdCPUAcc: Scenario

3

Executing other instructions



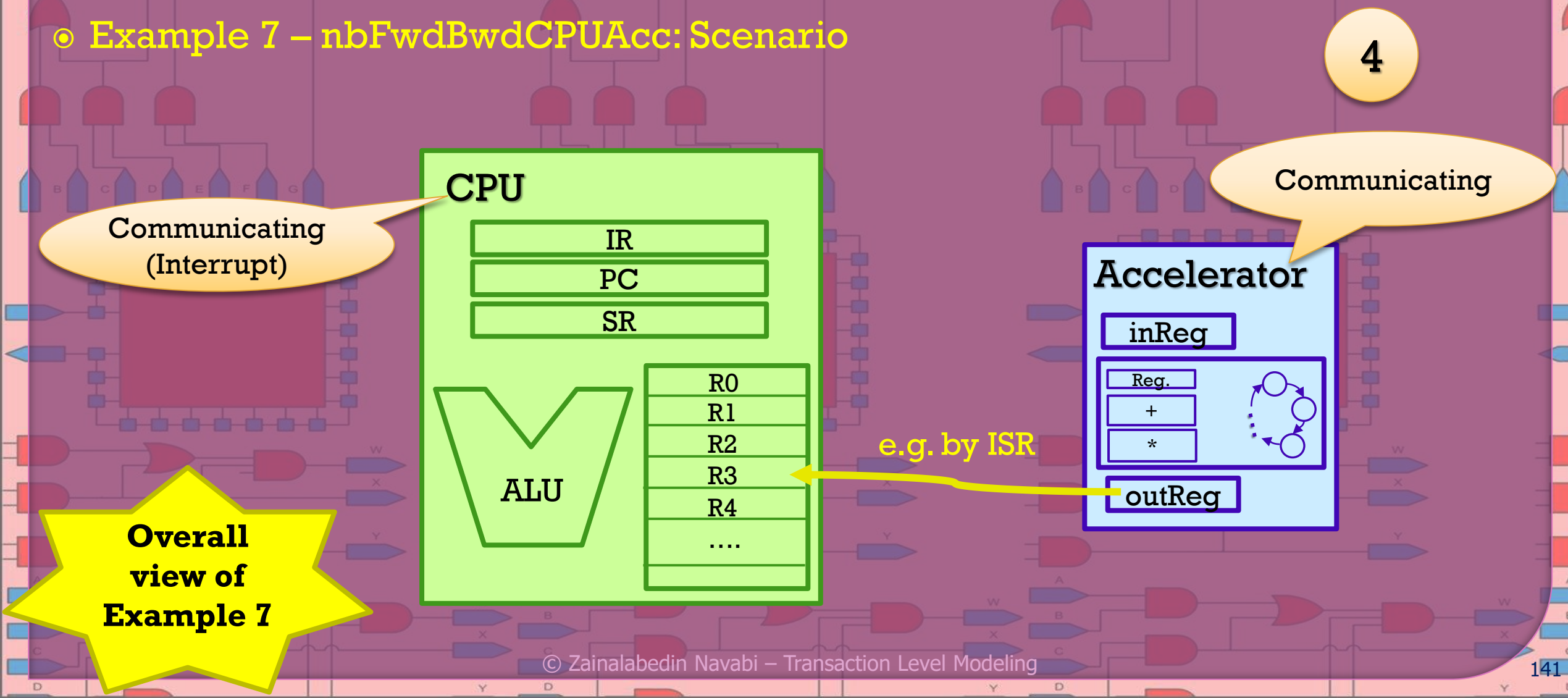
Calculating



Overall view of Example 7

Non-blocking Transport: Forward & Backward

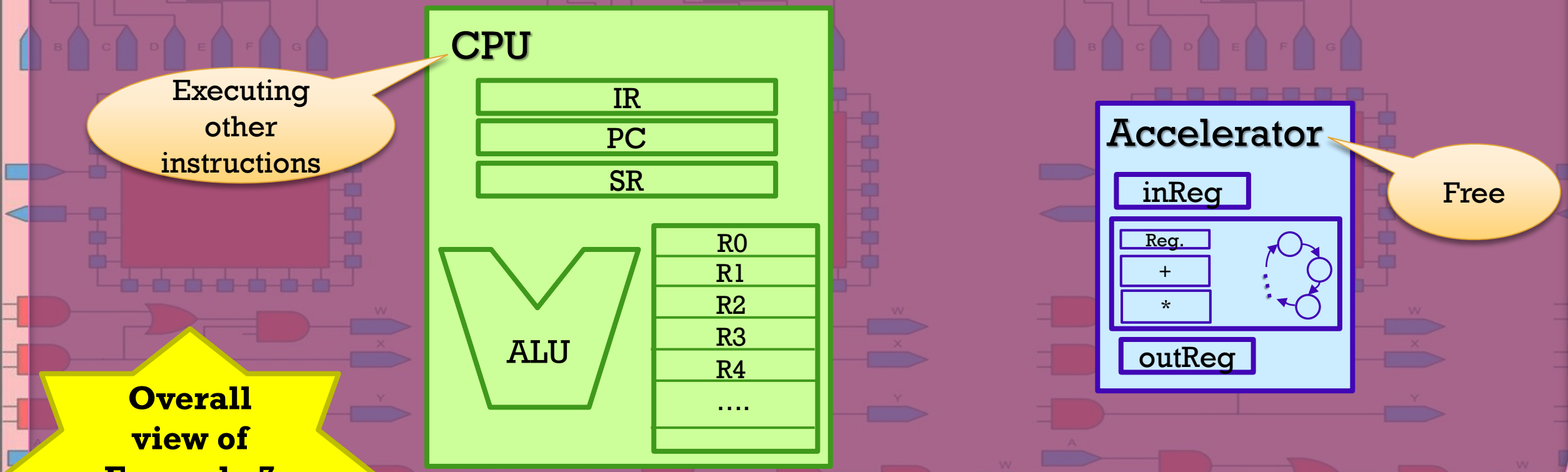
- Example 7 – nbFwdBwdCPUAcc: Scenario



Non-blocking Transport: Forward & Backward

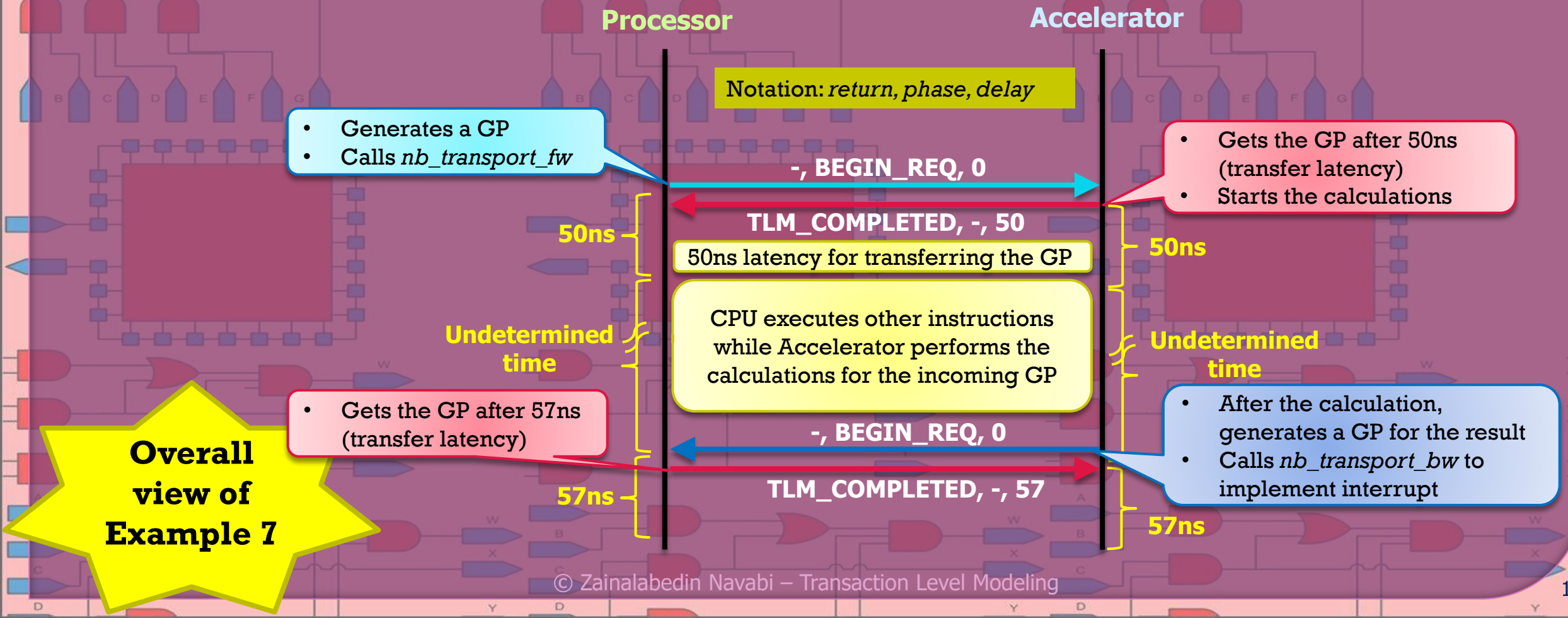
- Example 7 – nbFwdBwdCPUAcc: Scenario

5



Non-blocking Transport: Forward & Backward

⊙ Example 7 – nbFwdBwdCPUAcc: Scenario (More details)



Non-blocking Transport: Forward & Backward

- Example 7 – nbFwdBwdCPUAcc: Scenario
  - Processor (CPU) needs to perform intensive computations (for example calculating exponential or floating-point multiplication). It decides to entrust its computations to an accelerator (exponential accelerator that designed in Topic 4, or FP multiplier accelerator)
  - The processor generates a generic payload and calls *nb_transport_fw* to send the input data to the accelerator
  - The transfer latency from CPU to Accelerator has been considered 50ns, which includes the request accept delay, the latency of the target, and the response accept delay
  - The accelerator receives the input data after 50ns and then starts the calculations while the processor executes other instructions
  - When the calculations are done, the accelerator generates a generic payload and interrupts the CPU for the provided result by calling *nb_transport_bw* method
  - The transfer latency from CPU to Accelerator is 57ns
  - The processor receives the result after 57ns
  - ….

Interrupt creates a temporary halt during program execution and allows peripheral devices to access the processor. The processor responds to that interrupt with an Interrupt Service Routine (ISR)

Non-blocking Transport: Forward & Backward

- Example 7 – nbFwdBwdCPUAcc: Initiator, *Processor*

Processor.h | Processor.cpp | Accelerator.h | Accelerator.cpp

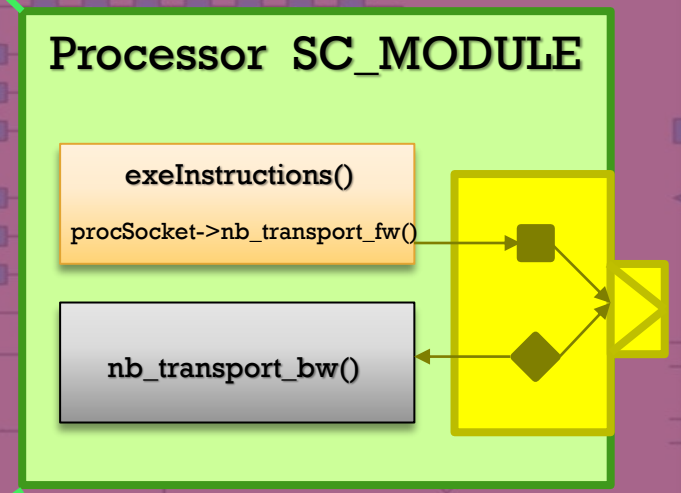
(Global Scope)

```
1  #include <systemc.h>
2  #include "tlm.h"
3  #include "tlm_utils/simple_initiator_socket.h"
4
5  class Processor : public sc_module {
6  public:
7      tlm_utils::simple_initiator_socket<Processor, 32> procSocket;
8
9      SC_CTOR(Processor) : procSocket("Processor_Socket"), inData(0)
10     {
11         inData = new tlm::tlm_generic_payload();
12         SC_THREAD(exeInstructions);
13         procSocket.register_nb_transport_bw(this, &Processor::nb_transport_bw);
14     }
15     tlm::tlm_generic_payload* inData;
16     void exeInstructions();
17     virtual tlm::tlm_sync_enum nb_transport_bw(tlm::tlm_generic_payload&, tlm::tlm_phase&, sc_time&);
18     int data;
19     int result;
20 };
```

Processor.h

Register callback for incoming *nb_transport_bw* interface method call

SC_THREAD for starting communication



Non-blocking Transport: Forward & Backward

Processor.h | Processor.cpp | Accelerator.h | Accelerator.cpp

Processor.cpp

```
#include "Processor.h"

void Processor::exeInstructions(){
    tlm::tlm_phase forwardPhase;
    sc_time delay;

    for (int i = 64; i < 104; i = i + 8){
        cout << "****************************************************************" << endl;
        tlm::tlm_command cmd = tlm::TLM_WRITE_COMMAND;
        if (cmd == tlm::TLM_WRITE_COMMAND)  data = i;

        inData->set_command( cmd );
        inData->set_address( 0 );
        inData->set_data_ptr( (unsigned char*) &data );
        inData->set_data_length( 1 );
        inData->set_streaming_width( 1 );
        inData->set_byte_enable_ptr( 0 );
        inData->set_dmi_allowed( false );
        inData->set_response_status( tlm::TLM_INCOMPLETE_RESPONSE );

        forwardPhase = tlm::BEGIN_REQ;
        delay = sc_time(0, SC_NS);
        cout << "The CPU sends an input data: " << (sc_lv<8>)data;
        cout << " at " << sc_time_stamp() << " delay=" << delay << '\n';
        tlm::tlm_sync_enum returnStatus;
        returnStatus = procSocket->nb_transport_fw(*inData, forwardPhase, delay);
        wait(delay);
        if (returnStatus == tlm::TLM_COMPLETED){
            if (inData->is_response_error())
                SC_REPORT_ERROR("TLM-2", "Error in memory handling of b_transport");
            cout << "The CPU is done with communication at " << sc_time_stamp() << endl;
        }
        cout << "The CPU is processing some instruction at " << sc_time_stamp() << '\n';
        wait(1300, SC_NS);
    }
}
```

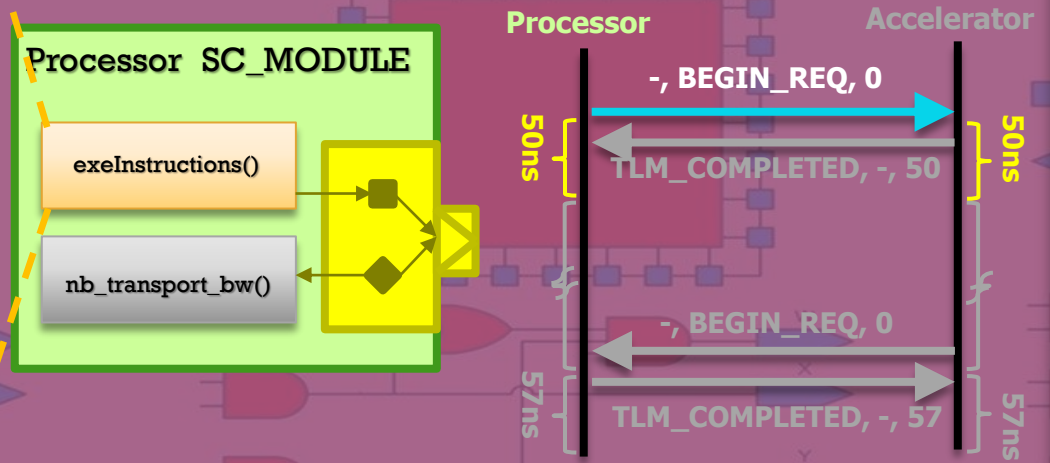
Generates a generic payload

To execute the current instruction (writing the input data in the accelerator)

Call *nb_transport_fw* to send the input data to the accelerator

For doing some instructions and for the interrupt time including 57ns for communication

- Example 7 – nbFwdBwdCPUAcc: Initiator, *Processor*



Non-blocking Transport: Forward & Backward

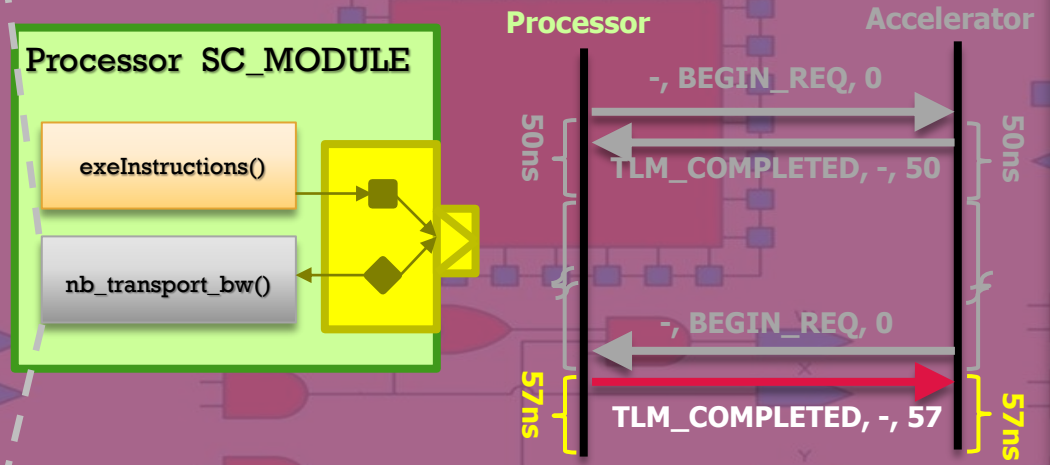
Processor.h | Processor.cpp | Accelerator.h | Accelerator.cpp

Processor.cpp

Processor

```
39  tlm::tlm_sync_enum Processor::nb_transport_bw(tlm::tlm_generic_payload& receivedTrans,
40      tlm::tlm_phase& phase, sc_time& delay){
41      tlm::tlm_command cmd = receivedTrans.get_command();
42      uint64           adr = receivedTrans.get_address();
43      unsigned char*   ptr = receivedTrans.get_data_ptr();
44      unsigned int     len = receivedTrans.get_data_length();
45      unsigned char*   byt = receivedTrans.get_byte_enable_ptr();
46      unsigned int     wid = receivedTrans.get_streaming_width();
47
48      if (byt != 0) {
49          receivedTrans.set_response_status(tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE);
50          return tlm::TLM_COMPLETED;
51      }
52      if (len != 1 || wid != len) {
53          receivedTrans.set_response_status(tlm::TLM_BURST_ERROR_RESPONSE);
54          return tlm::TLM_COMPLETED;
55      }
56
57      if (cmd == tlm::TLM_READ_COMMAND)
58              *(ptr) = *((unsigned char*)(&data + adr));
59
60      else if (cmd == tlm::TLM_WRITE_COMMAND)
61              *((unsigned char*)(&result + adr)) = *(ptr);
62
63      receivedTrans.set_response_status(tlm::TLM_OK_RESPONSE);
64      delay = delay + sc_time(57, SC_NS);
65      cout << "The result data: " << (sc_lv<8>)(*((unsigned char*)(&result + adr)));
66      cout << " was recieved by the CPU at " << sc_time_stamp() << endl;
67      return tlm::TLM_COMPLETED;
68  }
```

- Example 7 – nbFwdBwdCPUAcc: Initiator, *Processor*



Non-blocking Transport: Forward & Backward

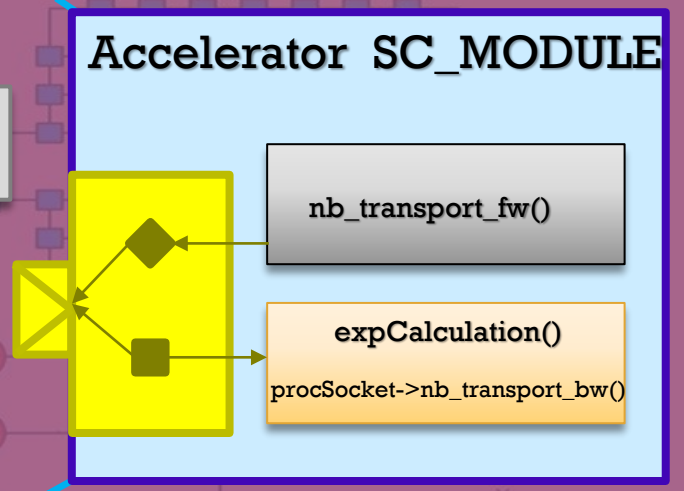
- Example 7 – nbFwdBwdCPUAcc: Target, *Accelerator*

Accelerator.h

```
Processor.h   Processor.cpp   Accelerator.h   Accelerat
(Global Scope)
 5  class Accelerator : public sc_module {
 6  public:
 7      tlm_utils::simple_target_socket<Accelerator, 32> accSocket;
 8
 9      SC_CTOR(Accelerator) : accSocket("Accelerator_socket") {
10
11          startCal = SC_LOGIC_0;
12
13          accSocket.register_nb_transport_fw(this, &Accelerator::nb_transport_fw);
14          SC_THREAD(expCalculation);
15          sensitive << startCal;
16      }
17
18      virtual tlm::tlm_sync_enum nb_transport_fw( tlm::tlm_generic_payload&,
19                                                  tlm::tlm_phase&, sc_time& );
20      void expCalculation();
21      int inReg;
22      int outReg;
23      sc_signal <sc_logic> startCal;
24  };
```

Register callback for incoming *nb_transport_fw* interface method call

Function for accelerator calculation and calling *nb_transport_fw* to interrupt the CPU for communicating back



Non-blocking Transport: Forward & Backward

Processor.h | Processor.cpp | Accelerator.h | Accelerator

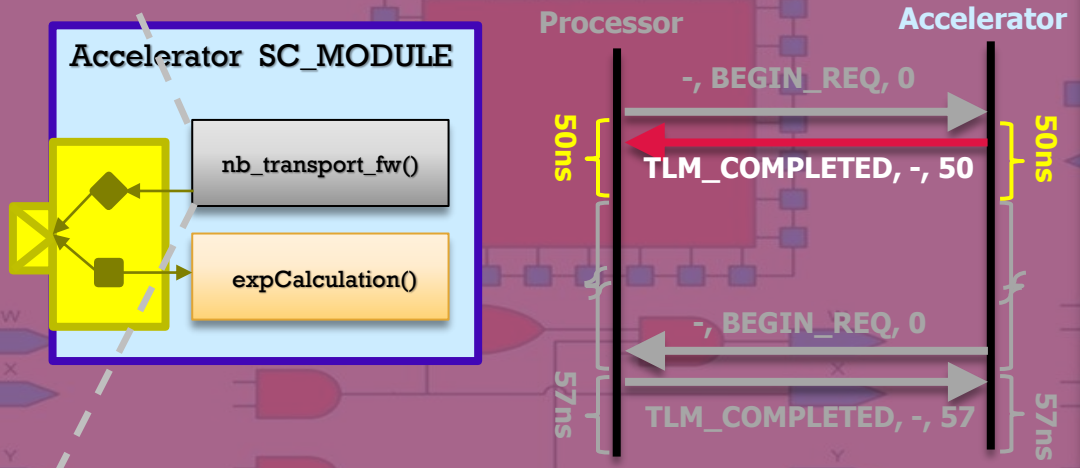
Accelerator

Accelerator.cpp

```
1  #include "Accelerator.h"
2
3  tlm::tlm_sync_enum Accelerator::nb_transport_fw
4        (tlm::tlm_generic_payload& receivedTrans,
5         tlm::tlm_phase& phase, sc_time& delay ){
6
7    tlm::tlm_command cmd = receivedTrans.get_command();
8    uint64           adr = receivedTrans.get_address();
9    unsigned char*   ptr = receivedTrans.get_data_ptr();
10   unsigned int     len = receivedTrans.get_data_length();
11   unsigned char*   byt = receivedTrans.get_byte_enable_ptr();
12   unsigned int     wid = receivedTrans.get_streaming_width();
13
14   if (byt != 0) {
15       receivedTrans.set_response_status( tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE );
16       return tlm::TLM_COMPLETED;
17   }
18   if (len != 1 || wid != len) {
19       receivedTrans.set_response_status( tlm::TLM_BURST_ERROR_RESPONSE );
20       return tlm::TLM_COMPLETED;
21   }
22
23   if ( cmd == tlm::TLM_READ_COMMAND )
24       *(ptr) = *((unsigned char*) (&outReg));
25   else if ( cmd == tlm::TLM_WRITE_COMMAND )
26       *((unsigned char*)(&inReg + adr)) = *(ptr);
27
28   startCal = ~startCal.read();
29   receivedTrans.set_response_status( tlm::TLM_OK_RESPONSE );
30   delay = delay + sc_time(50, SC_NS);
31   return tlm::TLM_COMPLETED;
32 }
```

To activate *expCalculation* function

- Example 7 – nbFwdBwdCPUAcc: Target, *Accelerator*



Non-blocking Transport: Forward & Backward

Accelerator.cpp

Transfer latency = 50 ns

Calculation time

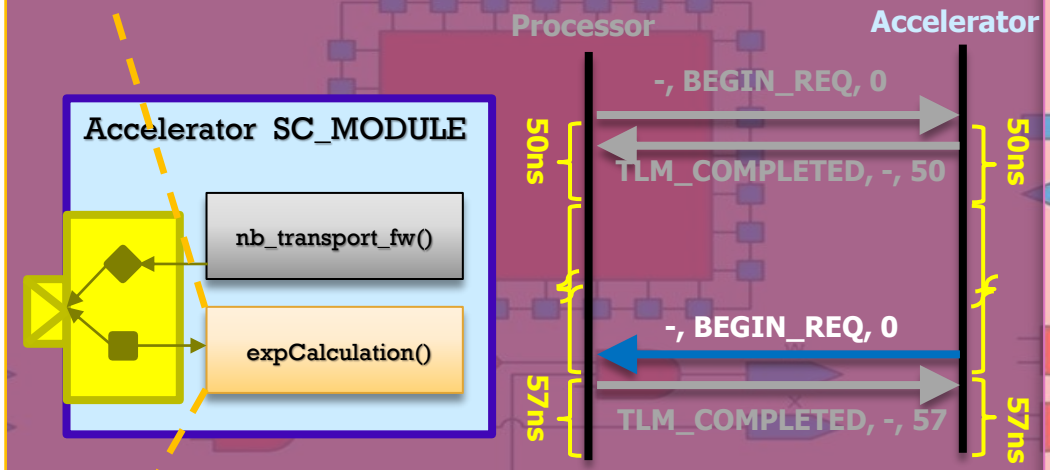
Generates a generic payload

Call *nb_transport_bw* to interrupt the processor for the provided result

Transfer latency = 57 ns

```
Processor.h  Processor.cpp  Accelerator.h  Accelerator.cpp
Accelerator                                nb_transport_fw(tlm::tlm_ge
34  void Accelerator::expCalculation(){
35      while (true){
36          wait(50, SC_NS);
37          cout << "The input data: " << (sc_lv<8>)inReg;
38          cout <<" was recieved by the accelerator at " << sc_time_stamp() << endl;
39          // do some calculations
40          cout << "The accelerator starts the calculations at " << sc_time_stamp() << endl;
41          wait((rand() % 200) + 400, SC_NS);
42
43          tlm::tlm_generic_payload* outData = new tlm::tlm_generic_payload;
44          tlm::tlm_command cmd = tlm::TLM_WRITE_COMMAND;
45          if (cmd == tlm::TLM_WRITE_COMMAND) outReg = rand();
46          outData->set_command(cmd);
47          outData->set_address(3);
48          outData->set_data_ptr((unsigned char*)&outReg);
49          outData->set_data_length(1);
50          outData->set_streaming_width(1);
51          outData->set_byte_enable_ptr(0);
52          outData->set_dmi_allowed(false);
53          outData->set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);
54
55          tlm::tlm_phase backwardPhase = tlm::BEGIN_REQ;
56          sc_time delay = sc_time(0, SC_NS);
57          cout << "The accelerator sends the result: " << (sc_lv<8>)outReg;
58          cout << " at " << sc_time_stamp() << " delay=" << delay << endl;
59          tlm::tlm_sync_enum returnStatus;
60          returnStatus = accSocket->nb_transport_bw(*outData, backwardPhase, delay);
61          wait(delay);
62          if (returnStatus == tlm::TLM_COMPLETED){
63              if (outData->is_response_error())
64                  SC_REPORT_ERROR("TLM-2", "Error in memory handling of b_transport");
65              cout << "The accelerator is done and ready for the next execution at " << sc_time_stamp() << endl;
66          }
67          wait();
68      }
69  }
```

- Example 7 – nbFwdBwdCPUAcc: Target, *Accelerator*



Non-blocking Transport: Forward & Backward

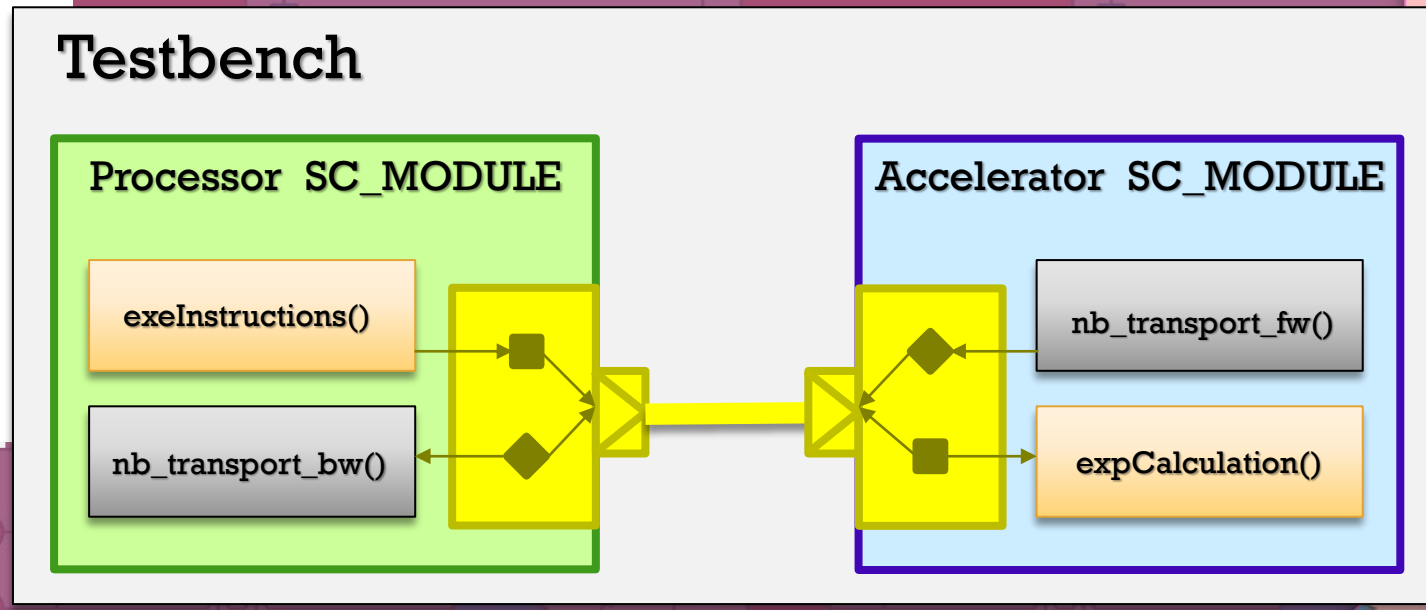
- Example 7 – nbFwdBwdCPUAcc: Testbench

Testbench.h

Testbench.h | Processor.h | Processor.cpp

(Global Scope)

```
1  #include "Processor.h"
2  #include "Accelerator.h"
3
4  SC_MODULE(Testbench)
5  {
6      Processor *CPU;
7      Accelerator *Acc;
8
9      SC_CTOR(Testbench)
10     {
11         CPU = new Processor("Processor");
12         Acc = new Accelerator("Accelerator");
13         CPU->procSocket.bind(Acc->accSocket);
14     }
15 };
```



Non-blocking Transport: Forward & Backward

- Example 7 – nbFwdBwdCPUAcc: Main

Main.cpp

Main.cpp | Testbench.h | Processor.h | Processor.cpp | Acce

(Global Scope)

```
1  #include "Testbench.h"
2
3  int sc_main(int argc, char* argv[])
4  {
5      Testbench TB("Testbench");
6      sc_start();
7      return 0;
8  }
```

Non-blocking Transport: Forward & Backward

⊙ Example 7 – nbFwdBwdCPUAcc: Result

```
C:\WINDOWS\system32\cmd.exe

        SystemC 2.3.1-Accellera --- Sep 29 2019 11:41:37
        Copyright (c) 1996-2014 by all Contributors,
        ALL RIGHTS RESERVED
**************************************************************
The CPU sends an input data: 01000000 at 0 s delay=0 s
The CPU is done with communication at 50 ns
The CPU is processing some instruction at 50 ns
The input data: 01000000 was recieved by the accelerator at 50 ns
The accelerator starts the calculations at 50 ns
The accelerator sends the result: 00100011 at 491 ns delay=0 s
The result data: 00100011 was recieved by the CPU at 491 ns
The accelerator is done and ready for the next execution at 548 ns
**************************************************************
The CPU sends an input data: 01001000 at 1350 ns delay=0 s
The CPU is done with communication at 1400 ns
The CPU is processing some instruction at 1400 ns
The input data: 01001000 was recieved by the accelerator at 1400 ns
The accelerator starts the calculations at 1400 ns
The accelerator sends the result: 10000100 at 1934 ns delay=0 s
The result data: 10000100 was recieved by the CPU at 1934 ns
The accelerator is done and ready for the next execution at 1991 ns
**************************************************************
```