2.4 Стандард TLM 2.0 - AT модели

Модели на AT нивоу абстракције користе неблокирајуће методе фундаменталних интерфејса. Комуникација се обавља вишеструким позивима неблокирајуох метода иницијаторског и циљног интерфејса, `nb_fw_transport` и `nb_bw_transport`. Трансакција се моделује стандардном генеричком трансакцијом (`tlm_generic_payload`). Комуникација пролази кроз више фаза од иницијације до завршетка. Да би компонента била комаптибилна са другим TLM компонентама потребно је поштовати основни протокол за комуникацију и четири фазе комуникације: почетак захтева (`BEGIN_REQ`), крај захтева (`END_REQ`), почетак одговора (`BEGIN_RESP`) и крај одговора (`END_RESP`).

Четири фазе комуникације садржане су у набројивом типу података `tlm_phase_enum`. Овај набројиви тип дефинисан је на овакав или сличан начин:

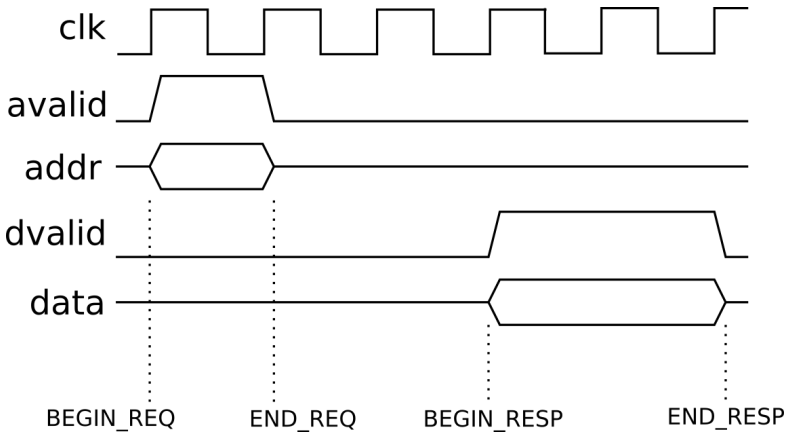
```
enum tlm_phase_enum { UNINITIALIZED_PHASE=0, BEGIN_REQ=1,
END_REQ, BEGIN_RESP, END_RESP };
```

Вредност `UNINITIALIZED_PHASE` постоји само да би се осигурало да фаза буде експлицитно иницијализована. Вредност `BEGIN_REQ` шаље се кроз иницијаторски прикључак. Комуникација започиње са овом фазом. Наредна фаза зависи од повратне вредности позива метода. Вредност `END_REQ` шаље се кроз циљни прикључак. Са овом фазом завршава се стање захтева основног протокола.

Вредност `BEGIN_RESP` шаље се кроз циљни прикључак. Наредна фаза зависи од повратне вредности. Вредност `END_RESP` шаље се кроз иницијаторски прикључак и са овом вредношћу завршава се стање одговора основног протокола.

Овим фазама се абстракује комуникација кроз комплексне магистрале. Један једноставан пример би била магистрала која се састоји од сигнала `avalid`, `addr`, `dvalid`, `data` (слика 2.18). Сигнал `avalid` би означавао да је вредност на адресном порту `addr` важећа. Адреса би након једног циклуса била прихваћена. Растућа ивица сигнала `avalid` почетак је захтева, а опадајућа ивица је крај захтева. Сигнал `dvalid` означава да су подаци на приступу за податке важећи. Активирање овог сигнала означава почетак одговора док деактивирање овог сигнала означава крај одговора.

Основни протокол је абстракован на овај начин зато што је то најопштија структура комуникације између две компоненте. Неке од наведених фаза по потреби могу бити и уклоњене приликом



Слика 2.18: Сигнали и фазе

имплементације комуникације. Да би се утврдило да ли се неке фазе основног протокола не користе, потребно је користити повратне вредности из неблокирајућих метода.

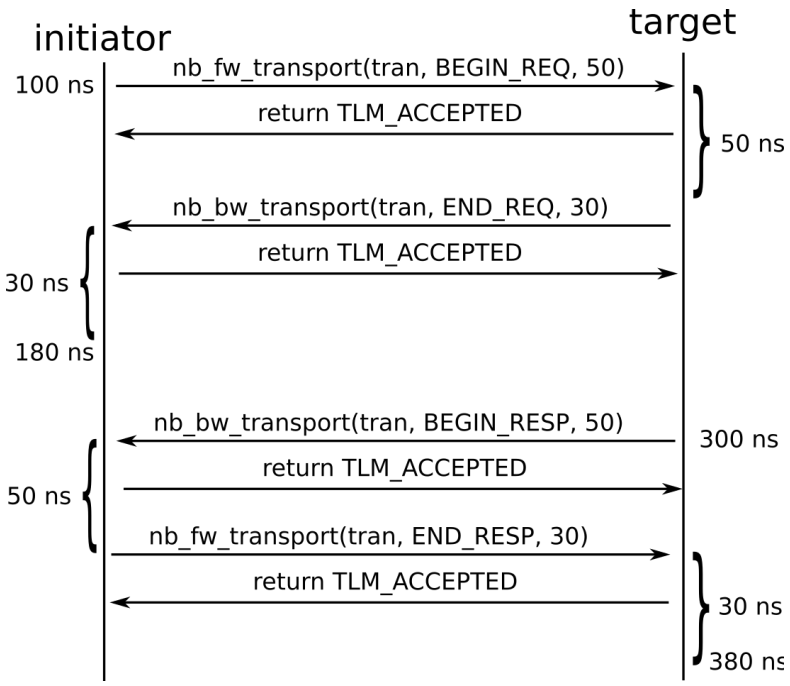
Повратне вредности неблокирајућих метода садржане су у набројивом типу података `tlm_sync_enum`, који може бити дефинисан на следећи начин:

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

Вредност `TLM_ACCEPTED` означава да се комуникација наставља по основном протоколу. Вредност `TLM_UPDATED` користи се да се прескоче неке фазе у комуникацији док се са `TLM_COMPLETED` означава да се комуникација завршава позивом те неблокирајуће методе. Комуникација која се одвија кроз све фазе основног протокола састоји се од 4 позива неблокирајућих метода од којих сваки позив као повратну вредност има `TLM_ACCEPTED`.

Један комплетан пример проласка кроз основни протокол састојао би се од проласка кроз све ове фазе (слика 2.19). Комуникација се започиње (100 ns), иницијаторским прикључком, позивом методе `nb_fw_transport` са фазом `BEGIN_REQ` и неким кашњењем. Циљна компонента, треба да моделује кашњење. Пошто се пролази кроз цео основни протокол, повратна вредност овог као и свих наредних позива је `TLM_ACCEPTED`. У тренутку 150 ns, циљни прикључак позива методу `nb_bw_transport` са фазом `END_REQ` и одговарајућим кашњењем (30 ns). Иницијаторска компоненте треба да поштује ово кашњење и да рачуна да је захтев усвојен тек у тренутку 180 ns.

Почетак одговора зависи од моделовања кашњења циљне компоненте и компоненти са којима она комуницира са својим осталим прикључцима.



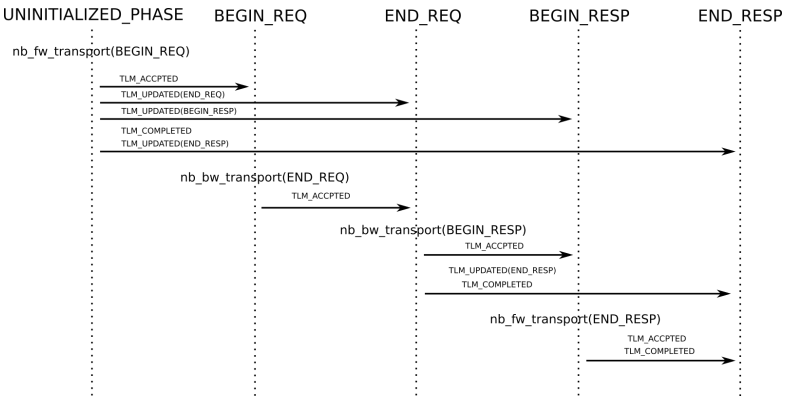
Слика 2.19: Фазе протокола

Након тог кашњења (кашење 120 ns, тренутак 300 ns) започиње се фаза одговора, позивом методе `nb_bw_transport` на циљном прикључку. Као фаза се прослеђује `BEG_RESP` и неко кашњење (50 ns). Иницијаторска компонента треба да моделује прослеђено кашњење и да позиве методу `nb_fw_transport` у одговарајућем тренутку (350 ns). Као фазу прослеђује `END_RESP` и кашњење (30 ns). Циљна компоненте након позива ове методе чека прослеђено време кашњења и након тог тренутка комуникација је завршена (380 ns).

Истовремено може да се одвија већи број трансакција на пљикључку. Но, само једна трансакција може да се прихвата на прикључку и само на једну трансакцију може да се шаље одговор. Ово ћемо звати правила екслузивног права прихвата и одговора. Правила су уведена стога што се на комплексним магистралама може одвијати само један захтев или одговор.

Основни протокол допушта само неке прелазе (слика 2.20). Комуникацију је могуће започети само позивом `nb_fw_transport` са параметром `BEGIN_REQ`. Након првог позива неблокирајуће методе `nb_fw_transport(BEGIN_REQ)` могући су прелази у све друге фазе комуникације у зависности од повратне вредности позива методе. Из фазе `BEGIN_REQ` могуће је прелаз само у фазу `END_REQ` позивом

`nb_fw_transport(END_REQ)`. Из фазе `END_REQ` могућ је позив само методе `nb_fw_transport` са параметром `BEGIN_RESP` и могући су прелази у фазе `BEGIN_RESP` и `END_RESP`, у зависности од повратне вредности. Док је фаза комуникације `BEGIN_RESP` могућ је само позив `nb_fw_transport(END_RESP)` и једини дозвољени прелаз је ка фази `END_RESP`.



Слика 2.20: Дозвољени прелази основног протокола

Имплементација AT модела је значајно комплекснија од имплементације LT модела. Уз све фазе транзиције које је потребно пратити, могуће је да већи број трансакција буде активан истовремено на компонентама, што је приближније реалној слици магистрала у физичком систему али се моделовање усложњава. Постоје и проблеми са праћењем које показивачи на трансакције су још увек важећи а да се прит томе симулазија не успори као и како ефикасно имплементирати захтевана кашњења транзиција. Постоје стандардне класе, које олакшавају имплементацију решења за неке од наведених проблема.

2.4.1 Редови догађаја трансакција

Да би се олакшала имплементација кашњења трансакција у фазама основног протокола, на располагању стоје класе за редове догађаја трансакција. Када се позове неблокирајућа метода са захтевом за прелаз на наредну фазу основног протокола након неког времена, повратна вредност мора да се врати без чекања. Проблем је како имплементирати кашњење ако имплементација не може да чека. Идеја је да се добијена трансакција убаци у ред догађаја који ће се активирати након траженог времена.

Стандардне класе за редове догађаја, Payload Event Queue (PEQ) постоје у TLM стандарду: `peq_with_get` и `peq_with_cb_and_phase`.

PEQ је класа која управља догађајима где је сваки догађај упарен са одговарајућом трансакцијом. Свака трансакција се уписује у PEQ заједно са својим догађајем и чита се из PEQ у тренутку који се срачуна као сума тренутног симулационог времена и прослеђеног времена кашњења. PEQ су концептуално значајни да би се разумело значење кашњења при имплементацији AT модела.

Најважније методе ових класа су:

```
namespace tlm_utils {

template <class PAYLOAD>
class peq_with_get
{
public:
  void notify(PAYLOAD& trans, const sc_core::sc_time& t);
  void notify(PAYLOAD& trans);
  PAYLOAD* get_next_transaction();
};

template<typename OWNER,
         typename TYPES=tlm::tlm_base_protocol_types>

class peq_with_cb_and_phase
{
public:
  typedef typename TYPES::tlm_payload_type
      tlm_payload_type;
  typedef typename TYPES::tlm_phase_type
      tlm_phase_type;
  typedef void (OWNER::*cb)(tlm_payload_type&,
      const tlm_phase_type&);

  peq_with_cb_and_phase(OWNER* _owner, cb _cb);
  void notify (tlm_payload_type& t, const tlm_phase_type& p,
      const sc_core::sc_time& when);
  void notify (tlm_payload_type& t, const tlm_phase_type& p);

}

}
```

Постоје две варијације методе `notify` у случају обе класе. Једна варијација има временски параметар као аргуемнт, а друга нема. У случају варијације без временског параметра, сматра се да је прослеђена вредност `SC_ZERO_TIME`. Временски параметар ове методе зваћемо циљано време.

Класа `peq_with_get` помоћу метода `notify` поставља пар догађај трансакција у ред. Догађај ће послати обавештење након циљаног времена. Класа која садржи PEQ трева да има процес који је осетљив на ова обавштења. Тај процес помоћу методе `get_next_transaction` може преузети трансакцију и у њему се имплементира основни протокол.

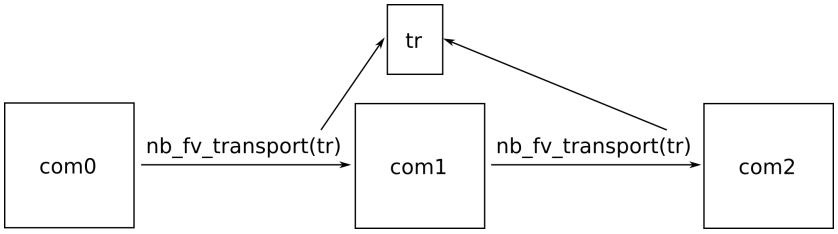
Контруктор класе `peq_with_cb_and_phase` прима као параметре показивач на класу и показивач на метод класе који мора бити тачно одговарајућег типа (`cb`). Ова метода је позивна и као параметре прима генеричку трансакцију и фазу комуникације. Метода notify прави пар догађај, трансакција који ставља у ред. Након циљаног времена позива се позивна метода класе са трансакцијом и фазом као параметрима. У позивној методи имеплементира се основни протокол.

Ове класе значајно олакшавају имплементацију AT модела. Неблокирајући транспортни позив треба да постави трансакцију у овај ред, док се имплементација основног протокола спроводи у одговарајћем процесу или позивној методи.

2.4.2 Управљање меморијом трансакција

Приликом моделовања AT модела, управљање животним веком трансакција представља проблем. Манипулација трансакцијама се обавља помоћу референци (слика 2.21). Процес који започиње комуникацију задужен је да обезбеди простор за трансакцију. Када се позове неблокирајућа метода већи број пута, већи број класа садржи референциу на исту трансакцију. Тренутак у коме је безбедно да се ослободи меморија није очигледан. Уколико се простор за трансакције не ослобађа, систем на коме се симулира модел остаће без меморију у случају довољно дуге симулације.

Да би AT модели имали добре перформасе симулације, потребно је имплементираи ефикасно управљање животним веком трансакција. Једно од генералних решења за управљање меморијом је помоћу бројања референци. Докле год је број референци на неки објекат већи од 0, тај објекат се задржава у моеморији. Када падне на нулу, објекат се уклања из меморије. Генеричка трансакција садржи методе `acquire` и `release`. Метода `acquire` повећава број референци, док метода `release` смањује број референци. Уз то метода `release`, уколико број референци



Слика 2.21: Проблем управљања меморијом

падне на 0, позива методу `free` система за управљање животним веком трансакција.

Стандард TLM стандардизује интерфејс класе која имплементира управљање животним веком трансакција, интерфејсом `tlm_mm_interface`. Овај интерфејс је једноставан:

```
class tlm_mm_interface {
public:
  virtual void free(tlm_generic_payload*) = 0;
  virtual ~tlm_mm_interface() {}
};
```

Да би класа имплементирала овај интерфејс потребно је да имплементира методу free коју позива метода release генеричке трансакције. Овај интерфејс може да се имплементира у посебној класи коју потом садржи иницијатор комуникације. Други начин је да сама класа која иницира комуникацију имплементира интерфејс.

2.4.3 Пример једноставног AT система

Пример AT система је исти као за LT систем (слика 2.17), али је начин имплементације потпуно другачији. Главни програм једноставног AT система истанцира и повезује иницијаторску и циљну компонентеу (листинг 2.8).

Листинг 2.8: Главни програм једноставног AT система

```
#include <systemc>
#include "generator.hpp"
#include "memory.hpp"

using namespace sc_core;
using namespace tlm;

int sc_main(int argc, char* argv[])
{
        memory mem ("memory_u");
        generator gen("generator_u");
        gen.isoc(mem.tsoc);
```

```
        sc_start(200, SC_NS);
    return 0;
}
```

Иницијатор комуникације имплементиран је класом generator. Ова класа налеђује иницијатоски интерфејс (листинг 2.9). Осим неопходног иницијаторског прикључка (`isoc`), као и метода иницијаторског интерфејса, ту је и процес (`gen`), који прави трансакције. Све је идентично као и са LT системом до сада. Промењљиве за директан приступ меморији нису коришћене у овом примеру. PEQ променљива је дефинисана (`m_peq`), као и систем за управљање животним веком трансакција (`mm`). За имплементацију екслузивног права прихвата коришћени су један догађај као и једна променљива (`req_in_progress` и `req_done`). Додатно, позивна метода којом се имплементира основни протокол је декларисана (`cb_peq`).

Листинг 2.9: Заглавље класе generator AT модел

```
#ifndef _GENERATOR_HPP_
#define _GENERATOR_HPP_

#include <systemc>
#include <tlm>
#include "mem_manager.hpp"
#include <tlm_utils/peq_with_cb_and_phase.h>

class generator :
        public sc_core::sc_module,
        public tlm::tlm_bw_transport_if<>
{
public:
        generator(sc_core::sc_module_name);

        tlm::tlm_initiator_socket<> isoc;

        typedef tlm::tlm_base_protocol_types::tlm_payload_type pl_t;
        typedef tlm::tlm_base_protocol_types::tlm_phase_type phase_t;

        tlm::tlm_sync_enum nb_transport_bw(pl_t&, phase_t&, sc_core::sc_time&);
        void invalidate_direct_mem_ptr(sc_dt::uint64, sc_dt::uint64);

protected:
        void gen();
        bool req_in_progess;
        sc_core::sc_event req_done;
        tlm_utils::peq_with_cb_and_phase<generator> m_peq;
        mem_manager mm;

        void cb_peq(tlm::tlm_generic_payload&, const tlm::tlm_phase&);
};

#endif
```

У конструктору класе региструје се процес и извози се сама класа у извозни део прикључка (листинг 2.10 `isoc`). PEQ прменљива се иницијализује са показивачем на иницијаторску класу (`this`) као и показивачем на позивну методу класе (`&generator::cb_peq`). Идентично као и у случају LT модела, главни процес (`gen`) у петљи, прави 20 трансакција за писање по меморији на сличајан начин а потом 20 трансакција читања и исписују се информативне поруке. Посотји и неколико значајних разлика у односу на LT систем.

Листинг 2.10: Имплементација класе generator AT модел

```
#include "generator.hpp"

using namespace sc_core;
using namespace sc_dt;
using namespace tlm;

SC_HAS_PROCESS(generator);

generator::generator(sc_module_name name) :
        sc_module(name),
        isoc("isoc"),
        req_in_progess(false),
        m_peq(this, &generator::cb_peq)
{
        SC_THREAD(gen);
        isoc(*this);
}


void generator::gen()
{
        phase_t phase;
        sc_time delay = sc_time(4, SC_NS);
        tlm_sync_enum rsp;
        unsigned plnum = 0;

        for (unsigned int i = 0; i != 40; ++i)
        {
                unsigned int data_length = 1 + rand() % 4;
                unsigned int addr = rand() % 200;
                tlm_command cmd = i < 20 ? TLM_WRITE_COMMAND : TLM_READ_COMMAND;

                pl_t* pl = mm.alloc();
                unsigned char* data = pl->get_data_ptr();

                for (unsigned i = 0; i != data_length; ++i)
                        data[i] = rand() % 100;

                pl->set_command          ( cmd                          );
                pl->set_address          ( addr                         );
                pl->set_data_length      ( data_length                  );
                pl->set_response_status ( TLM_INCOMPLETE_RESPONSE );
                pl->acquire();

                if(req_in_progess)
                        wait(req_done);
                req_in_progess = true;

                plnum++;
                phase = BEGIN_REQ;
                std::string msg = "Send " + std::to_string(plnum) +
                        " payload to address " + std::to_string(addr);
                SC_REPORT_INFO("generator", msg.c_str());

                rsp = isoc->nb_transport_fw(*pl, phase, delay);

                assert(rsp == TLM_ACCEPTED && pl->get_response_status() == TLM_OK_RESPONSE);
        }
}

tlm_sync_enum generator::nb_transport_bw(pl_t& pl, phase_t& phase, sc_time& delay)
{
        assert(phase == END_REQ || phase == BEGIN_RESP);
        m_peq.notify(pl, phase);

        return TLM_ACCEPTED;
}

void generator::cb_peq(tlm_generic_payload& pl, const tlm_phase& phase)
{
        switch(phase)
        {
        case END_REQ:
        {
                SC_REPORT_INFO("generator", "Request accepted.");
                req_in_progess = false;
                req_done.notify();
                return;
        }
        case BEGIN_RESP:
        {
```

```
                tlm_phase ret_phase = END_RESP;
                sc_time delay(1, SC_NS);
                isoc->nb_transport_fw(pl, ret_phase, delay);
                return;
        }
        default:
                SC_REPORT_FATAL("generator", "Bad phase");
        }
}

void generator::invalidate_direct_mem_ptr(uint64 start, uint64 end)
{

}
```

Прва разлика је начин на који се добија генеричка трансакција. У AT моделу, трансакција се добија методом `alloc` система за управљање трансакцијама. Бафер за смештање података за трансакцију се добија од генеричке трансакције. Систем за управљање трансакцијама, прави овај бафер и поставља га трансакцији. Друга разлика је што се позива метода `acquire` генеричке трансакције. Трећа разлика је постојање имплементације за екслузивно право прихвата. Уколико је нека трансакција у фази прихватања, процес чека док се прихватање не заврши. И наравно, последња разлика је позив неблокирајуће методе иницијаторског прикључка, уместо позива блокирајуће методе.

Метода за неблокирајућу комуникацију иницијаторске компоненте, `nb_transport_bw` користи PEQ за једноставнију имплементацију. Проверава се да ли је примљена фаза по основном протоколу и придошла трансакција се ставља у PEQ.

Позивна метода (`cb_peq`) имплементира основни протокол. Уколико је тренутна фаза `END_REQ`, ослобађа се екслузивно право на приступ. У случају да је фаза `BEGIN_RESP`, позива се метода циљног прикључка са наредном фазом `END_RESP` и кашњењем од 1 ns.

Метода `invalidate_direct_mem_ptr` је тривијално имплементирана пошто се не користи.

Циљна компонента имплементирана је класом memory која наслеђује циљни интерфејс (листинг 2.11). Слично као и код LT модела, ова класа садржи промењљиву која моделује низ локација меморије (`ram`). Као и код иницијаторске компоненте, дефинисана је PEQ променљива (`m_peq`), позивна метода (`cb_peq`) као и променљива и догађај који служе за имплементацију екслузивног права на одговор.

Листинг 2.11: Заглавље класе memory AT модел

```
#ifndef _MEMORY_HPP_
#define _MEMORY_HPP_

#include <systemc>
#include <tlm>
#include <tlm_utils/peq_with_cb_and_phase.h>

class memory :
        public sc_core::sc_module,
        public tlm::tlm_fw_transport_if<>
{
```

```
public:
        memory(sc_core::sc_module_name);

        tlm::tlm_target_socket<> tsoc;

        typedef tlm::tlm_base_protocol_types::tlm_payload_type pl_t;
        typedef tlm::tlm_base_protocol_types::tlm_phase_type phase_t;

        void b_transport(pl_t&, sc_core::sc_time&);
        tlm::tlm_sync_enum nb_transport_fw(pl_t&, phase_t&, sc_core::sc_time&);
        bool get_direct_mem_ptr(pl_t&, tlm::tlm_dmi&);
        unsigned int transport_dbg(pl_t&);

protected:
        static const int RAM_SIZE = 1024;
        unsigned char ram[RAM_SIZE];
        tlm_utils::peq_with_cb_and_phase<memory> m_peq;

        bool resp_in_progess;
        sc_core::sc_event resp_done;

        void cb_peq(tlm::tlm_generic_payload&, const tlm::tlm_phase&);
};

#endif
```

Конструктор циљне компоненте иницијализује све промењиве (листинг 2.12). Између осталих, PEQ се иницијализује показивачима коју омогућавају приступ позивној методи (`this`, `&memory::cb_req`). Класа се извози у извозни део циљног прикључка и низ који модлује локације меморије се иницијализује.

Имплементације блокирајуће методе `b_transport`, методе за директан приступ меморији `get_direct_mem_ptr`, као и методи за лов на грешке `transport_dbg` су тривијалне и нису у употреби у овом једноставном примеру.

Листинг 2.12: Имплементација класе memory AT модел

```
#include "memory.hpp"

using namespace sc_core;
using namespace tlm;
using namespace sc_dt;

memory::memory(sc_module_name name) :
        sc_module(name),
        tsoc("tsoc"),
        m_peq(this, &memory::cb_peq),
        resp_in_progess(false)
{
        tsoc(*this);
        for (int i = 0; i != RAM_SIZE; ++i)
                ram[i] = 0;
}

void memory::b_transport(pl_t& pl, sc_time& delay)
{

}

tlm_sync_enum memory::nb_transport_fw(pl_t& pl, phase_t& phase, sc_time& delay)
{
        m_peq.notify(pl, phase, delay);
        pl.set_response_status(TLM_OK_RESPONSE);
        return TLM_ACCEPTED;
}

void memory::cb_peq(tlm_generic_payload& pl, const tlm_phase& phase)
{
        switch(phase)
        {
        case BEGIN_REQ:
        {
```

```
                tlm_command cmd    = pl.get_command();
                uint64 adr         = pl.get_address();
                unsigned char *buf = pl.get_data_ptr();
                unsigned int len   = pl.get_data_length();
                std::string msg;

                switch(cmd)
                {
                case TLM_WRITE_COMMAND:
                        for (unsigned int i = 0; i != len; ++i)
                                ram[adr++] = buf[i];
                        msg = "Write ";
                        break;
                case TLM_READ_COMMAND:
                        for (unsigned int i = 0; i != len; ++i)
                                buf[i] = ram[adr++];
                        msg = "Read ";
                        break;
                default:
                        pl.set_response_status( TLM_COMMAND_ERROR_RESPONSE );
                }

                msg += " at time " + sc_time_stamp().to_string();
                msg += " to address " + std::to_string(adr) + " :\n";
                for (int i = 0; i != 200; ++i)
                {
                        msg += std::to_string(ram[i]);
                        msg += " ";
                }

                SC_REPORT_INFO("memory status", msg.c_str());

                sc_time delay(5, SC_NS);
                tlm_phase ret_phase = END_REQ;
                tlm_sync_enum rsp = tsoc->nb_transport_bw(pl, ret_phase, delay);

                assert(rsp == TLM_ACCEPTED);
                sc_time fw_delay(10, SC_NS);
                tlm_phase fw_phase = BEGIN_RESP;
                m_peq.notify(pl, fw_phase, fw_delay);

                break;

        }

        case BEGIN_RESP:
        {
                if(resp_in_progess)
                        wait(resp_done);
                resp_in_progess = true;
                sc_time delay(3, SC_NS);
                tlm_phase fw_phase = BEGIN_RESP;
                tsoc->nb_transport_bw(pl, fw_phase, delay);
                break;
        }

        case END_RESP:
        {
                resp_in_progess = false;
                resp_done.notify();
                SC_REPORT_INFO("memory", "Transaction finished.");
                pl.release();
                break;
        }

        default:
                SC_REPORT_FATAL("memory", "Bad phase");
        }
}

bool memory::get_direct_mem_ptr(pl_t& pl, tlm_dmi& dmi)
{
        return false;
}

unsigned int memory::transport_dbg(pl_t& pl)
{
        return 0;
}
```

Метода за неблокирајућу комуникацију `nb_transport_fw` циљног прикључка је једноставна зато што се користи PEQ. Улазна трансакција се ставља у PEQ са одговарајућим кашњењем. Поставља се да је одговор коректан и враћа се повратна вредност `TLM_ACCEPTED`. Сва комплексност имплементације, пребачена је у позивну методу.

Позивна метода (`cb_peq`) садржи три главна дела, у зависности од тренутне фазе трансакције. Уколико је фаза `BEGIN_REQ`, моделује се писање односно читање из меморије. Потом се позива неблокирајућа метода за комуникацију циљног прикључка са наредном фазом, `END_REQ`. На крају, у PEQ се ставља догађај са фазом `BEGIN_RES` који се активира након наведеног кашњења. Ово кашњење моделује време потребно меморији за обраду трансакције.

Када је фаза `BEGIN_RESP`, чека се ослобођење екслузивног права за одговор. Потом се преузима то право и позива се неблокирајућа метода циљног прикључка. У случају фазе `END_RESP`, ослобађа се екслузивно право и позива се метода release генеричке трансакције.

Преостало је још да се опише начин рада система за управљање животним веком трансакције. Овај систем имплементиран је класом `mem_manager` (листинг 2.13). Класа наслеђује интерфејс `tlm_mm_interface` и имплементира методу `free` овог интерфејса као и деструктор. Ту је и метода `alloc`, која није део овог интерфејса, којом се преузимају генеричке трансакције. Додатно, ту је и ред генеричких трансакција (`pls`) у коме се чувају већ направљене трансакције за поновно коришћење.

Листинг 2.13: Заглавље класе mem_manager

```
#ifndef _MEM_MANAGER_HPP_
#define _MEM_MANAGER_HPP_

#include <tlm>
#include <deque>

class mem_manager :
        public tlm::tlm_mm_interface
{
        typedef tlm::tlm_base_protocol_types::tlm_payload_type pl_t;
public:

        pl_t* alloc();
        void free(pl_t*);
        ~mem_manager();

protected:
        std::deque<pl_t*> pls;
};

#endif
```

Метода `alloc` проверава да ли постоје већ направљене генеричке трансакције и уколико постоје преузиме једну од њих и враћа показивач на ту трансакцију (листинг lst:mmcpp). На овај начин позива се оператор new само уколико је то неопходно, а то је у слулају када је ред празан.

Тада се прави нова генеричка трансакција са бафером великим 100 бајтова.

Листинг 2.14: Имплементација класе mem_manager

```
#include "mem_manager.hpp"

using namespace std;
using namespace tlm;

mem_manager::pl_t* mem_manager::alloc()
{
        pl_t* p_pl;

        if(pls.empty())
        {
                p_pl = new pl_t(this);
                unsigned char* data = new unsigned char [100];
                p_pl->set_data_ptr(data);
        }
        else
        {
                p_pl = pls.front();
                pls.pop_front();
                SC_REPORT_INFO("Manager", "reuse!");
        }

        return p_pl;
}

void mem_manager::free(pl_t* p_pl)
{
        p_pl->reset();
        pls.push_back(p_pl);
}

mem_manager::~mem_manager()
{
        for (size_t i = 0; i != pls.size(); ++i)
        {
                delete [] pls[i]->get_data_ptr();
                delete pls[i];
        }
        pls.clear();
}
```

Метода `free` ресетује генеричку трансакцију, што значи да јој поставља бројач референци на почетну вредност. Потом чува генеричу трансакцију у реду, за евентуално поновно коришћење. Деструктор пролази кроз све трансакције у реду и брише њихове бафере као и саме трансакције.

Овај једноставан пример илуструје пролазак кроз основни протокол. Но, овај пример није потпуно компатибилан са TLM стандардом. Да би модел био једноставнији нису проверавани сви могући случајеви транзиције (слика 2.20), што би компатибилан модел трабао да имплементира. Уз илустрацију основног протокола, коришћења PEQ класе, приказан је и једноставан систем за управљање животним веком трансакција.

2.4.4 Вежбе

1. Моделовати неки IIR филтар 5 реда, TLM AT стилом. Модел треба да садржи регистар у који се уписује вредност улазног одбирка,

као и регисар који садржи вредност излазног одбирка. Направити тестбенч за овај модел. Тестбенч треба да садржи референтни модел са којим се пореди TLM - AT модел.

2. Моделовати генератор Фибоначијевих бројава TLM AT стилом. Модел треба да садржи улазни регистар који одређује колико је дугачка генерисана секвенца. Постоји и регисар који садржи вредност генерисаног броја. Направити тестбенч за овај модел. Тестбенч треба да садржи референтни модел са којим се пореди TLM - AT модел.