2.5 Виртуелна платформа

Виртуелне платформе, VP, су софтверски модели чија сврха је да буду замена хардверским прототиповима у раним фазама развоја система. Софтверски модели битних делова система се повезују са циљем да се добије модел целокупног система који може да се извршава. Један од циљева развоја TLM стандарда је био да се олакша прављење виртуелних платформи.

Приликом развоја виртуелних платформи потребно је направити компромис између прецизности модела и брзине његовог извршавања. Модели треба да садрже довољно детаља да се софтвер може извршавати на њима без грешки, али требају да буду и довољно абстрактни да би били довољно брзи и омогућили ефикасну верификацију.

Типични VP модели користе моделе процесора прецизне у циклус заједно са моделима меморија као и кључних периферија. Целокупан модел треба да је довољно прецизан, тако да коначна верзија софтвера (бинарне датотеке) може без модификација да се извршава на VP моделу као и на хардверском прототипу. Гледано са стране развоја софтвера, VP модел не сме да се разликује од хардверског прототипа.

Предности виртуелних платформи у односу на хардверске прототипове могу бити:

- Рани развој
- Видљивост и контрола
- Доступност
- Перформансе

Виртуелне платформе омогућавају ранији развој софтвера. VP модели су доступни знатно раније од хардверских прототипова у развојном циклусу, што омогућава ранији почетак развоја софтверских компоненти система, а самим тим и ранију појаву коначног производа на тржишту.

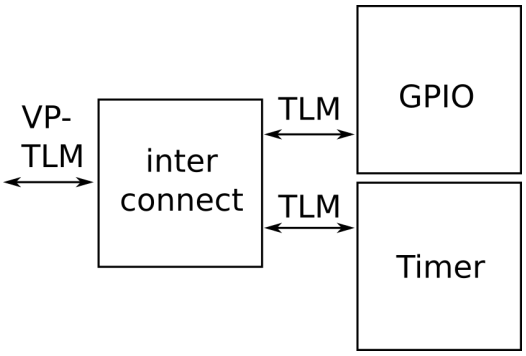
VP модели пружају потпуну видљивост у начин рада целокупног система. Додатно ови модели омогућавају да се једноставно измене параметри свих делова система што олакшава контролу. Хардверски прототипови садрже ограничен приступ само неким интерним регистрима и сигналима и не постоји једноставан начин да се ови параметри мењају. Због потпуне видљивости и контроле, могу се направити добри алати који омогућавају ефикасну верификацију.

VP модели широко су доступни развојном тиму. Модели могу бити доступни на свим рачунарским платформама унутар развојног тима, за разлику од хардверских прототипова чија доступност је веома ограничена у раним фазама развоја. Доступност омогућава већем броју инжињера да раде истовремено на развоју софтвера.

VP модели могу имати веома добре перформансе. Уколико су пажљиво моделовани, VP модели, у неким случајевима, могу радити брже од хардверских прототипова. Ово је могуће због коришћења абстракције и стога што се модели симулирају на рачунарима опште намене који су обично веома брзи. Хардверски прототипови често имају процесор ограничених могућности због ограничења коначног производа. У овом случају, VP омогућава додатно време за развој и верификацију.

2.5.1 Меморијско мапирање

Главни разлог настанка TLM стандарда био је да се омогући моделовање VP компоненти које могу да се користе у различитим алатима, различитих фирми. TLM интерфејс је стандардни интерфејс за моделовање виртуелних платформи. Једноставан VP модел може да се састоји од једног мерача времена и једне General Purpose Input Output (GPIO) компоненте (слика 2.22).



Слика 2.22: Једноставна виртуелна платформа

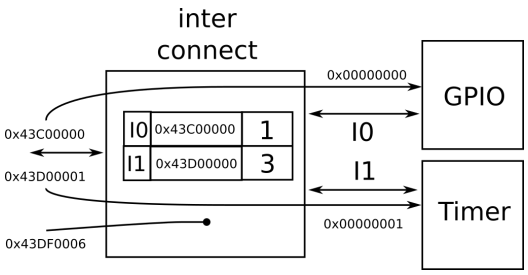
Да би се ове две компоненте повезале у јединствен систем потребно је да имају један, заједнички приступ (VP-TLM). Остатак система, који може бити Central Processing Unit (CPU), слао би све трансакције на овај приступ. Компонента која повезује већи број различитих компоненти и абстрахује их једним приступом назива се интерконект (interconnect).

Механизам којим интерконект компонента сакрива остале компоненте иза само једног приступа назива се меморијско мапирање.

Идеја меморијског мапирања је једноставна и може се исказати на следећи начин: поступај према свим компонентама система као да се обичне локације у јединственој меморији. То значи да свака компонента система добија јединствену адресу. Начин придруживања адреса компонентама система може да се уради на велики број начина. Битно је да свака компонента има јединствену адресу. Ове адресе се називају још и базне адресе компоненит. Нека базна адреса мерача времена буде 0x43D00000, а нека адреса GPIO компоненте буде 0x43C00000.

Свака од компоненти може имати свој меморијски подсистем. На пример, унутар компоненте може постојати већи број конфигурационих регистара или меморија за обраду података. Меморијски подсистем компоненте има своје локалне адресе. Адреса меморијског подсистема у јединственом адресном простору добија се сабирањем базне и локалне адресе. На пример, нека мерач времена садржи три регистра и нека су њихове локалне адресе 0, 1 и 2. Адресе ова три регистра у јединственом адресном простору су 0x43D00000, 0x43D00001 и 0x43D00002.

Интерконект је компонента која пребацује адресу у јединственом адресном простору у локалну адресу одговарајуће компоненте и обрнуто. Интерконект садржи табелу у којој је за сваки излазни интерфејс наведена базна адреса као и опсег важећих локалних адреса (слика 2.23). На пример, интерконект за једноставну виртуелну платформу (слика 2.22) садржи две врсте, по једну за сваки излазни интерфејс. За интерфејс који води ка GPIO компоненти (`I0`) постављена је базна адреса 0x43C00000 а за вредност важећих локалних адреса наведен је број 1, пошто ова компонента садржи само један регистар. Други интерфејс је повезан на мерач времена (`I1`), па је постављена базна адреса 0x43D00000 а важећи опсег је 3, стога што ова компонента садржи три регистра.



Слика 2.23: Превођење адреса

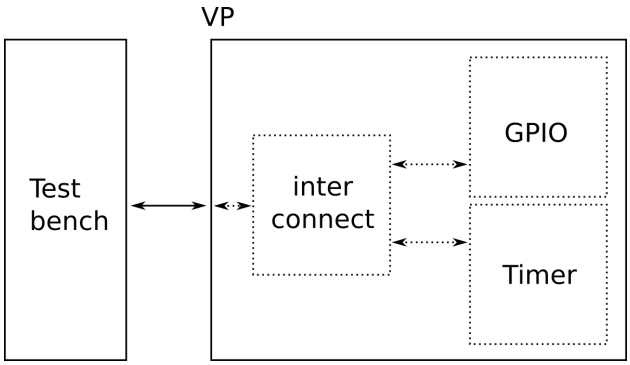
Постоје три могуће ситуације, за једноставну виртуелну платформу. Уколико се на јединственом улазном интерфејсу појави трансакција која адресира 0x43C00000, она ће бити прослеђена GPIO компоненти. Уколико стигне трансакција која као адресу садржи вредност у опсегу

[0x43D00000, 0x43D00002], она ће бити прослеђена мерачу времена. Ако на интерконект стигне трансакција са било којом другом адресом, интерконект компонента је неће проследити и евентуално ће пријавити грешку.

За трансакције са валидним адресама, интерконект од јединствене адресе одузима базну адресу и прослеђује трансакцију на одговарајћу интерфејс са локалном адресом. Трансакција са промењеном адресом се прослеђује одговарајућој компоненти. На пример, уколико пристигне трансакција са адресом, 0x43D00001, интерконект ће на основу табеле разумети да је трансакција намењена мерачу времена. Од јединствене адресе, одузеће се базна адреса и на тај начин добиће се локална адреса за мерач времена. Трансакција са адресом 0x00000001 проследиће се мерачу времена. Превођење адреса се одвија и када компоненте шаљу трансакције. Тада се на локалну адресу додаје базна адреса и на тај начин се добија адреса у јединственом адресном простору. И у овом случају трансакција са преведеном адресом се прослеђује на јединствени интерфејс виртуелне платформе.

2.5.2 Модел једноставне виртуелне платформе

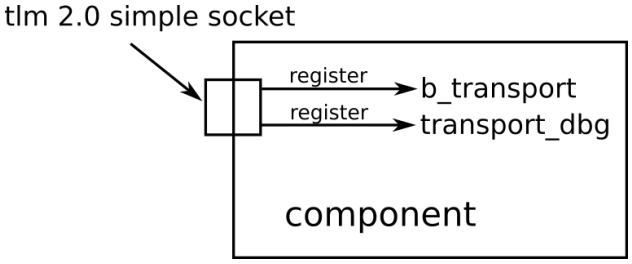
Модел једноставне виртуелне платформе имплементиран је коришћењем стандардних TLM интерфејса. Пројектован је и једноставан тестбенч, да би се провериле основне могућности система (слика 2.24). Платформа је моделована LT стилом, стога ни у једном модулу не користи се метода `wait`.



Слика 2.24: Тестбенч за VP модел

До сада је за имплементацију TLM компоненти коришћено наслеђивање иницијаторског и циљног интерфејса. У овом моделу, биће представљен алтернативни начин имплементације TLM интерфејса.

У до сада приказаним примерима, компонента је наслеђивала фундаменталне интерфејсе, имаплементирала их и у свом конструктору се повезивала на извозни део TLM прикључка. У овом примеру, главна компонента садржи помоћну компоненту која већ имплементира фундаменталне интерфејсе (слика 2.25). Помоћна компонента омогућује да се региструју позивне функције или методе помоћу којих се мења начин комуникације. Постоји већи број ових једноставних помоћних прикључака. У овом примеру користиће се `simple_target_socket` и `simple_initiator_socket`. Главна компонента, користи помоћну компоненту за комуникацију тако што региструје своје позивне методе. Предност алтернативног начина имаплементације је у том што главна компонента садржи само методе у којима је потребна нека функционалност. На овај начин избачене су методе које су празне али морају да се имплементирају да би модел био семантички исправан. Празне имеплементације свих метода интерфејса се налазе у помоћним компонентама.



Слика 2.25: Једноставни прикључци

Заглавље компоненте gpio садржи дефиницију модула који садржи прикључак `simple_target_socket` (листинг 2.15). Да би се омогућила употреба једноставног прикључка (`soc`) потребно је укључити заглавље `tlm_utils/simple_target_socket.h`. Модул садржи и вредност интерног регистра (`val`), позивну методу `b_transport`, као и методу за испис трансакције (`msg`).

Листинг 2.15: Заглавље gpio модула

```
#ifndef _GPIO_H_
#define _GPIO_H_

#include <tlm>
#include <tlm_utils/simple_target_socket.h>

class gpio :
        public sc_core::sc_module
{
public:
        gpio(sc_core::sc_module_name);

        tlm_utils::simple_target_socket<gpio> soc;
```

```
protected:
        sc_dt::sc_uint<8> val;

        typedef tlm::tlm_base_protocol_types::tlm_payload_type pl_t;
        void b_transport(pl_t&, sc_core::sc_time&);
        void msg(const pl_t&);
};

#endif
```

Конструктор `gpio` модула региструје позивну методу `b_transport` у прикључку `soc` (листинг 2.16). Позивна метода `b_transport` уписује вредност у регистар `val`, када је у питању трансакција писања. У случају трансакције читања, вредност регистра се ставља као вредност на показивач података трансакције. На улазно време се додаје кашњење, пошто је у питању LT модел. Метода `msg` исписује трансакцију и информације битне за `gpio` модул помоћу `SC_REPORT_INFO` макроа.

Листинг 2.16: Имплементација gpio модула

```
#include "gpio.hpp"
#include <tlm>

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

gpio::gpio(sc_module_name name):
        sc_module(name),
        soc("soc")
{
        soc.register_b_transport(this, &gpio::b_transport);
}

void gpio::b_transport(pl_t& pl, sc_time& offset)
{
        tlm_command    cmd  = pl.get_command();
        uint64         addr = pl.get_address();
        unsigned char* data = pl.get_data_ptr();

        if (addr < 0x10)
        {
                switch(cmd)
                {
                case TLM_WRITE_COMMAND:
                {
                        val = *((sc_uint<8>*)pl.get_data_ptr());
                        pl.set_response_status( TLM_OK_RESPONSE );
                        msg(pl);
                        break;
                }
                case TLM_READ_COMMAND:
                {
                        *data = (unsigned char) val;
                        pl.set_response_status( TLM_OK_RESPONSE );
                        msg(pl);
                        break;
                }
                default:
                        pl.set_response_status( TLM_COMMAND_ERROR_RESPONSE );
                        SC_REPORT_ERROR("GPIO", "TLM bad command");
                }
        }
        else
                SC_REPORT_WARNING("GPIO", "TLM wrong address");
        offset += sc_time(3, SC_NS);
}

void gpio::msg(const pl_t& pl)
{
        stringstream ss;
        ss << hex << pl.get_address();
        sc_uint<8> val = *((sc_uint<8>*)pl.get_data_ptr());
        string cmd  = pl.get_command() == TLM_READ_COMMAND ? "read  " : "write ";
```

```
        string msg = cmd + "val: " + to_string((int)val) + " adr: " + ss.str();
        msg += " @ " + sc_time_stamp().to_string();

        SC_REPORT_INFO("GPIO", msg.c_str());
}
```

Компонента мерача времена имплементирана је модулом `timer` (листинг 2.17). Модул `timer` комплекснији је од модула `gpio`. И овај модул садржи једноставни помоћни прикључак (`soc`). Овај модул треба да мери време у дискретним временским интервалима. Величину интервала могуће је подесити методом `set_period`. Величина интервала чува се у интерној променљивој (`period`). Мерач времена садржи три регистра. Регистар `cfg` садржи два индикатора. Индикатор на нижој позицији означава да ли мерач треба да ради, док индикатор на вишој позицији, означава да ли је истекло задато време. Време које треба да се мери поставља се у регистру `cnt_reload`. Тренутна вредност мереног времена налази се у регистру `cnt`. Локалне адресе ових регистата излистане су у заглављу: `cfg` је на адреси 0, `cnt` је на адреси 1 а `cnt_reload` је на адреси 2. Позивна метода (`b_transport`) као о метода за испис порука (`msg`) су декларисани. Додатно овај модул садржи и процес (`proc`), који мери протекло време. У заглављу налази се и имеплементација методе `set_period`.

Листинг 2.17: Заглавље timer модула

```
#ifndef _TIMER_H_
#define _TIMER_H_

#include <tlm>
#include <tlm_utils/simple_target_socket.h>

const sc_dt::uint64 TIMER_CFG = 0;
const sc_dt::uint64 TIMER_CNT = 1;
const sc_dt::uint64 TIMER_RLD = 2;

class timer :
        public sc_core::sc_module
{
public:
        timer(sc_core::sc_module_name);

        tlm_utils::simple_target_socket<timer> soc;

        inline void set_period(sc_core::sc_time&);

protected:
        sc_core::sc_time period;
        sc_dt::sc_uint<2> cfg;
        sc_dt::sc_uint<32> cnt;
        sc_dt::sc_uint<32> cnt_reload;
        typedef tlm::tlm_base_protocol_types::tlm_payload_type pl_t;

        void b_transport(pl_t&, sc_core::sc_time&);
        void proc();
        void msg(const pl_t&);
};

void timer::set_period(sc_core::sc_time& t)
{
        period = t;
}

#endif
```

Модул `timer` је интересантан пошто је прва циљна компонента која садржи процес (листинг 2.18). То је назначено макроом `SC_HAS_PROCESS`. Овај макро може се налазизи и у имплементацији. У конструктору се региструје позивна метода и региструје се и процес. Позивна метода `b_transport` уписује или чита вредности из регистара `cfg`, `cnt` и `cnt_reload` у зависности од адресе трансакције. На улазно време додаје се фиксно кашњење чиме се моделује протоко времена током транспорта у LT стилу. Метода `msg` исписује трансакције на начин погодан за мерач времена.

Листинг 2.18: Имплементација timer модула

```
#include "timer.hpp"
#include <tlm>
#include <tlm_utils/tlm_quantumkeeper.h>

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

SC_HAS_PROCESS(timer);

timer::timer(sc_module_name name):
        sc_module(name),
        soc("soc"),
        period(1, SC_NS),
        cfg(0)
{
        soc.register_b_transport(this, &timer::b_transport);
        SC_THREAD(proc);
}

void timer::b_transport(pl_t& pl, sc_time& offset)
{
        tlm_command    cmd  = pl.get_command();
        uint64         addr = pl.get_address();
        unsigned char* data = pl.get_data_ptr();

        switch(cmd)
        {
        case TLM_WRITE_COMMAND:
        {
                switch(addr)
                {
                case TIMER_CFG:
                        cfg = *((sc_uint<2>*)data);
                        if (cfg & 0x1)
                                cnt = cnt_reload;
                        pl.set_response_status( TLM_OK_RESPONSE );
                        break;
                case TIMER_CNT:
                        cnt = *((sc_uint<32>*)data);
                        pl.set_response_status( TLM_OK_RESPONSE );
                        break;
                case TIMER_RLD:
                        cnt_reload = *((sc_uint<32>*)data);
                        pl.set_response_status( TLM_OK_RESPONSE );
                        break;
                default:
                        pl.set_response_status( TLM_ADDRESS_ERROR_RESPONSE );
                        break;
                }
                break;
        }
        case TLM_READ_COMMAND:
        {
                switch(addr)
                {
                case TIMER_CFG:
                        memcpy(data, &cfg, sizeof(cfg));
                        pl.set_response_status( TLM_OK_RESPONSE );
                        break;
                case TIMER_CNT:
```

```
                        memcpy(data, &cnt, sizeof(cnt));
                        pl.set_response_status( TLM_OK_RESPONSE );
                        break;
                case TIMER_RLD:
                        memcpy(data, &cnt_reload, sizeof(cnt_reload));
                        pl.set_response_status( TLM_OK_RESPONSE );
                        break;
                default:
                        cout << "TIMER bad address.\n";
                        pl.set_response_status( TLM_ADDRESS_ERROR_RESPONSE );
                        break;
                }
                break;
        }
        default:
                pl.set_response_status( TLM_COMMAND_ERROR_RESPONSE );
                SC_REPORT_ERROR("TIMER", "TLM bad command");
                break;
        }

        msg(pl);
        offset += sc_time(4, SC_NS);
}

void timer::msg(const pl_t& pl)
{
        stringstream ss;
        ss << hex << pl.get_address();
        sc_uint<32> val = *((sc_uint<32>*)pl.get_data_ptr());
        string cmd  = pl.get_command() == TLM_READ_COMMAND ? "read  " : "write ";

        string regname;
        switch(pl.get_address())
        {
        case 0: regname = "CFG"; break;
        case 1: regname = "CNT"; break;
        case 2: regname = "RLD"; break;
        default: regname = "no reg";
        }

        string msg = cmd + "val: " + to_string((int)val) + " adr: " + ss.str();
        msg += " " + regname;
        msg += " @ " + sc_time_stamp().to_string();

        SC_REPORT_INFO("TIMER", msg.c_str());
}

void timer::proc()
{
        tlm_utils::tlm_quantumkeeper qk;
        qk.reset();

        while(1)
        {
                if(cfg & 0x1)
                {
                        cnt--;
                        if(cnt == 0)
                        {
                                cfg |= 0x2;
                                cnt = cnt_reload;
                                SC_REPORT_INFO("TIMER", "Counter expired");
                        }
                }
                qk.inc(period);
                if (qk.need_sync())
                {
                        qk.sync();
                        SC_REPORT_INFO("TIMER", string(
                                                        "Synced @ " +
                                                        sc_time_stamp().to_string()).c_str());
                }
        }
}
```

Модул timer је циљни модул који садржи процес `proc`. Уз то ова класа је модолевана LT стилом, односно коришћено је временско раздвајање. Стога је у имплементацији овог процеса коришћена класа `tlm_quantumkeeper` (`qk`). Овај процес садржи бесконачну петљу у којој

се проверава да ли време треба да се мери. У случају да треба, смањује се регистар `cnt` и када вредност овог регистра досегне вредност 0, одговарајући индикатор регистра `cfg` се поставља. Сваки пролаз кроз петљу увећава вредност локалног времена за вредност дискретног корака, `qk.inc(period)`. Када вредност локалног времена постане веће од глобалног квантног времена долази до синхронизације, `qk.sync()`. У том тренутку, овај процес предаје контролу кернелу симулатора.

Ово је типичан пример LT модела. Симулација се значајо убрзава, зато што овај модул не предаје контролу кернелу симулатора након сваког смањења бројача, као што би био случај код модела прецизног у циклус. Уместо тога контрола се пребацује тек након што се досегне глобално квантно време. За моделовање времена не користи се метода `wait`, већ се користи временско раздвајање и класа `tlm_quantumkeeper`.

Интерконект модул имплементиран је модулом `sys_bus` (листинг 2.19), који садржи два једноставна иницијаторска помоћна прикључка (`s_gpio`, `s_timer`) и један једноставан циљни помоћни прикључак (`c_cpu`). Иницијаторски прикључци су предвиђени да се повежу са GPIO компонентом и мерачом времена, док је циљни прикључак предвиђен за повезивање са остатком система. У овом примеру то ће бити тестбенч, а у другом примеру то ће бити процесор. Позивна метода `b_transport` декларисана је, као и метода за испис порука, `msg`.

Листинг 2.19: Заглавље sys_bus модула

```
#ifndef _SYS_BUS_HPP_
#define _SYS_BUS_HPP_

#include <systemc>
#include <tlm>
#include <tlm_utils/simple_target_socket.h>
#include <tlm_utils/simple_initiator_socket.h>

class sys_bus :
        public sc_core::sc_module
{
public:
        sys_bus(sc_core::sc_module_name);

        tlm_utils::simple_target_socket<sys_bus> s_cpu;
        tlm_utils::simple_initiator_socket<sys_bus> s_gpio;
        tlm_utils::simple_initiator_socket<sys_bus> s_timer;

protected:
        typedef tlm::tlm_base_protocol_types::tlm_payload_type pl_t;
        void b_transport(pl_t&, sc_core::sc_time&);
        void msg(const pl_t&);
};

#endif
```

Конструктор модула `sys_bus` (листинг 2.20) региструје позивну методу `b_transport` на циљни помоћни прикључак. Унутар имплементације позивне методе `b_transport` транслирају се адресе из јединственог адресног простора у локалне адресе компоненти. Уместо бројева, за адресе се користе константе (листинг 2.21). Пре позива

транспортне методе иницијаторских прикључака, адреса се мења са јединствене на локалну. Пошто се транспортна метода заврши, на крају позивне методе `b_transport`, модула `sys_bus`, поново се поставља јединствена адреса. На овај начин транслирање адреса нема утицаја на компоненту која је иницирала транспорт података. На крају, ту је и имеплементација методе за испис трансакција.

Листинг 2.20: Имплементација sys_bus модула

```
#include "sys_bus.hpp"
#include "vp_addr.hpp"
#include <string>

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

sys_bus::sys_bus(sc_module_name name) :
        sc_module(name)
{
        s_cpu.register_b_transport(this, &sys_bus::b_transport);
}

void sys_bus::b_transport(pl_t& pl, sc_core::sc_time& offset)
{
        uint64 addr = pl.get_address();
        uint64 taddr;
        offset += sc_time(2, SC_NS);

        if(addr == VP_ADDR_GPIO)
        {
                taddr = addr & 0x0000000F;
                pl.set_address(taddr);
                s_gpio->b_transport(pl, offset);
                msg(pl);
        }
        else if (addr >= VP_ADDR_TIMER &&
                        addr <  VP_ADDR_TIMER_H)
        {
                taddr = addr & 0x0000000F;
                pl.set_address(taddr);
                s_timer->b_transport(pl, offset);
                msg(pl);
        }

        pl.set_address(addr);
}

void sys_bus::msg(const pl_t& pl)
{
        stringstream ss;
        ss << hex << pl.get_address();
        sc_uint<32> val = *((sc_uint<32>*)pl.get_data_ptr());
        string cmd  = pl.get_command() == TLM_READ_COMMAND ? "read  " : "write ";

        string msg = cmd + "val: " + to_string((int)val) + " adr: " + ss.str();
        msg += " @ " + sc_time_stamp().to_string();

        SC_REPORT_INFO("BUS_FWD", msg.c_str());
}
```

Листинг 2.21: Дефиниције адреса за виртуелну платформу

```
#ifndef _VP_TSP_ADDR_H_
#define _VP_TSP_ADDR_H_

#include "timer.hpp"

const sc_dt::uint64 VP_ADDR_GPIO = 0x43C00000;
const sc_dt::uint64 VP_ADDR_TIMER = 0x43D00000;
const sc_dt::uint64 VP_ADDR_TIMER_CFG = VP_ADDR_TIMER + TIMER_CFG;
const sc_dt::uint64 VP_ADDR_TIMER_CNT = VP_ADDR_TIMER + TIMER_CNT;
const sc_dt::uint64 VP_ADDR_TIMER_RLD = VP_ADDR_TIMER + TIMER_RLD;
const sc_dt::uint64 VP_ADDR_TIMER_H = 0x43D00004;
```

```
#endif
```

Виртуелна платформа имплементирана је модулом `vp` (листинг 2.22). Овај модул садржи један једноставан циљни прикључак (`s_cpu`), који је јаван. Овај прикључак служи да се виртуелна платформа повеже са остатком система. Уз то, модул садржи и један једноставан иницијаторски прикључак који је заштићен (`s_bus`) и служи да пребаци трансакције са улазног јавног прикључка на интерконект модул. Сви саставни делови виртуелне платфоме су статички дефинисану унутар модула (`sb`, `gp`, `t`). И овај модул садржи позивну методу `b_transport`.

Листинг 2.22: Заглавље vp модула

```
#ifndef _VP_HPP_
#define _VP_HPP_

#include <systemc>
#include <tlm_utils/simple_target_socket.h>
#include <tlm_utils/simple_initiator_socket.h>
#include "sys_bus.hpp"
#include "gpio.hpp"
#include "timer.hpp"

class vp :
        sc_core::sc_module
{
public:
        vp(sc_core::sc_module_name);

        tlm_utils::simple_target_socket<vp> s_cpu;

protected:
        tlm_utils::simple_initiator_socket<vp> s_bus;
        sys_bus sb;
        gpio gp;
        timer t;

        typedef tlm::tlm_base_protocol_types::tlm_payload_type pl_t;
        void b_transport(pl_t&, sc_core::sc_time&);
};

#endif
```

Имплементација `vp` модула је једноставна (листинг 2.23). Унутар конструктора повезане су све компоненте помоћу `bind` методе једноставних прикључака. Додатно је регистрована позивна метода `b_transport` на улазном јавном прикључку (`s_cpu`). Позивна метода `b_transport` само прослеђује трансакције пристигле са јавног прикључка на заштићени прикључак.

Листинг 2.23: Имплементација vp модула

```
#include "vp.hpp"
#include <iostream>

using namespace sc_core;

vp::vp(sc_module_name name) :
        sc_module(name),
        sb("sys_bus"),
        gp("gpio"),
        t("timer")
{
        s_cpu.register_b_transport(this, &vp::b_transport);

        s_bus.bind(sb.s_cpu);
```

```
        sb.s_gpio.bind(gp.soc);
        sb.s_timer.bind(t.soc);
        SC_REPORT_INFO("VP", "Platform is constructed");
}

void vp::b_transport(pl_t& pl, sc_time& delay)
{
        s_bus->b_transport(pl, delay);
        SC_REPORT_INFO("VP", "Transaction passes...");
}
```

Модул `tb_vp` имплементира тестбенч за виртуелну платформу (листинг 2.24). Унутар модуча налате се дефинисани један једноставан иницијаторски прикључак (`isoc`), један процес (`test`) као и метода за испис трансакција `msg`.

Листинг 2.24: Заглавље tb_vp модула

```
#ifndef _TB_VP_H_
#define _TB_VP_H_

#include <tlm_utils/simple_initiator_socket.h>

class tb_vp :
        public sc_core::sc_module
{
public:
        tb_vp(sc_core::sc_module_name);

        tlm_utils::simple_initiator_socket<tb_vp> isoc;

protected:

        void test();

        typedef tlm::tlm_base_protocol_types::tlm_payload_type pl_t;

        void msg(const pl_t&);
};


#endif
```

Конструктор модула `tb_vp` региструје процес `test` (листинг 2.25). Процес `test` садржи једноставне команде чији је циљ да се тестирају основне могћности виртуелне платформе. Пошто се тестирају могућности модела који имплементира временско раздвајање, и овај процес садржи класу `tlm_quantumkeeper`. Прво се унутар петље тестира GPIO модул. Затим се конфигурише мерач времена. Поставља се интервал времена који се мери, уписом одговарајуће вредности у регистар `RLD`. Потом се покреће мерач времена постављањем одговарајућег индикатора, унутар регистра `CFG`. Потом се унутар петље читају вредности из регистра `CFG` мерача времена и проверава се да ли је мерач времена одбројао задати интервал. Када се утврди да јесте, петља се прекида. Након 1 μs, симулација се завршава позивом `sc_stop`. На крају, ту је и имплементација методе за испис трансакција `msg`.

Листинг 2.25: Имплементација tb_vp модула

```
#include "tb_vp.hpp"
#include "vp_addr.hpp"
#include <string>
#include <tlm_utils/tlm_quantumkeeper.h>
```

```
using namespace sc_core;
using namespace sc_dt;
using namespace std;
using namespace tlm;

SC_HAS_PROCESS(tb_vp);

tb_vp::tb_vp(sc_module_name name):
        sc_module(name)
{
        SC_THREAD(test);
}

void tb_vp::test()
{
        sc_time loct;
        tlm_generic_payload pl;
        tlm_utils::tlm_quantumkeeper qk;
        qk.reset();

        //************************************************************************
        // Test GPIO every 5 ns
        //************************************************************************
        for (int i = 0; i != 10; ++i)
        {
                sc_uint<8> val = i+1;
                pl.set_address(VP_ADDR_GPIO);
                pl.set_command(TLM_WRITE_COMMAND);
                pl.set_data_length(1);
                pl.set_data_ptr((unsigned char*)&val);

                isoc->b_transport(pl, loct);
                qk.set_and_sync(loct);
                msg(pl);

                loct += sc_time(5, SC_NS);
        }

        //************************************************************************
        // Configure timer
        //************************************************************************
        sc_uint<32> val = 0x00001000;
        pl.set_address(VP_ADDR_TIMER_RLD);
        pl.set_command(TLM_WRITE_COMMAND);
        pl.set_data_length(1);
        pl.set_data_ptr((unsigned char*)&val);
        isoc->b_transport(pl, loct);
        msg(pl);
        qk.set_and_sync(loct);

        sc_uint<2> cfg = 1;
        pl.set_address(VP_ADDR_TIMER_CFG);
        pl.set_command(TLM_WRITE_COMMAND);
        pl.set_data_length(1);
        pl.set_data_ptr((unsigned char*)&cfg);
        isoc->b_transport(pl, loct);
        msg(pl);
        qk.set_and_sync(loct);

        //************************************************************************
        // Wait until timer is done.
        //************************************************************************
        while(1)
        {
                sc_uint<32> tmp;
                pl.set_address(VP_ADDR_TIMER_CFG);
                pl.set_command(TLM_READ_COMMAND);
                pl.set_data_length(1);
                pl.set_data_ptr((unsigned char*)&tmp);
                isoc->b_transport(pl, loct);
                msg(pl);
                qk.set_and_sync(loct);

                if (tmp & 0x2) break;
                else
                {
                        qk.set_and_sync(qk.get_local_time() + sc_time(200, SC_NS));
                        loct = qk.get_local_time();
                        SC_REPORT_INFO("TB", string("Synced @ " +
                                                    sc_time_stamp().to_string()).c_str());
                }
        }
```

```
        //*************************************************************************
        // Wait 1 more micro second and finish simulation.
        //*************************************************************************
        qk.inc(sc_time(1, SC_US));
        qk.sync();
        sc_stop();
}

void tb_vp::msg(const pl_t& pl)
{
        static int trcnt = 0;
        stringstream ss;
        ss << hex << pl.get_address();
        sc_uint<32> val = *((sc_uint<32>*)pl.get_data_ptr());

        string msg = "val: " + to_string((int)val) + " adr: " + ss.str();
        msg += " @ " + sc_time_stamp().to_string();

        SC_REPORT_INFO("TB", msg.c_str());
        trcnt++;
        SC_REPORT_INFO("TB", ("------------" + to_string(trcnt)).c_str());
}
```

Главни програм модела, унутар `sc_main` функције, инстанцира виртуелну платформу и тестбенч и потом их повезује (листинг 2.26). Затим се поставља глобално квантно време и покреће се симулација.

Листинг 2.26: Главни програм за једноставну виртуелну платформу

```
#include <systemc>
#include "tb_vp.hpp"
#include "vp.hpp"

using namespace sc_core;

int sc_main(int argc, char* argv[])
{
        vp uut("uut");
        tb_vp tb("tb_vp");
        tb.isoc.bind(uut.s_cpu);

        tlm::tlm_global_quantum::instance().set(sc_time(100, SC_NS));

        sc_start();

    return 0;
}
```

Главна добит виртуелних платформи је могућност ранијег почетка развоја софтвера. Да би се ова могућност искористила, софтвер се развија на алатима који могу да симулирају процесор. Примери таквих алата су ISS и емулатори. У једном од наредних поглавља биће коришћен Quick EMUlator (QEMU) да се напише једноставан софтверски модул који одсликава могућности ове платформе.

2.5.3 Вежбе

1. Развити скалабилну интерконект компоненту која може да се користи за развој TLM виртуелних платформи. Компонента треба да саржти један улазни TLM прикључак и произвољан број излазних прикључака. Компонента треба да садржи табелу помоћу које се меморијски мапира, као и методу `memmap`, помоћу које се

мења та табела. Развити и једноставан тестбенч, који демонстрира употребу интерконект компоненте.

2. Моделовати виртуелну платформу система, која ће садржати следеће LT компоненте: GPIO, мерач времена и IIR филтар. За виртуалну платофму, потребно је развити и тестбенч, који ће демонстрирати основне могућности система.

3. Моделовати виртуелну платформу система, која ће садржати следеће LT компоненте: GPIO, мерач времена и генератор Фибоначијевих бројава. За виртуалну платофму, потребно је развити и тестбенч, који ће демонстрирати основне могућности система.