2.1 Симулација модела развијених у различитим језицима

Приликом пројектовања комплексних система на чипу, готово никад се не имплементира цео дизајн од почетка. Углавном, постоји већи број готових IP блокова, спремних за поновно коришћење или постоје целе готове платформе. Ови наслеђени IP блокови су обично развијани у неком од HDL језика. За развој архитектуре система користи се SystemC језик због своје флексибилности при моделовању. Стога, да би се успешно спровео развој комплексног електронског система, неопходни су симулатори који могу да симулирају HDL и SystemC моделе, истовремено. OSCI симулатор нема ову могућност и стога му је употребна вредност донекле ограничена. Комерцијални симулатори могу да симулирају системе развијане у различитим језицима и то је главни разлог зашто се они користе у великим пројектима. У наставку текста подразумеваће се употреба Xcelium симулатора.

2.1.1 Укључивање HDL модела у SystemC моделе

Приликом инстацирања компоненте у SystemC моделу, која је писана у неком од HDL језика, потребно је написати или генерисати SystemC модул, који ће представити HDL модел као да је у питању SystemC модул. То се постиже наслеђивањем класе `sc_foreign_module`.

Унутар имплементације класе, која неслеђује `sc_foreign_module`, наводе се сви портови које HDL модел користи, и наводи се један конструктор у који се као параметри прослеђују име инстанце унутар SystemC модела, као и име HDL модула, односно ентитета, који се инстанцира. Додатно, потребно је имплементирати методу `hdl_name`, базне класе `sc_foreign_module`, чија повратна вредност је стринг који ће се користити као име модула у оквиру симулатора.

Шаблон за имплементацију класе омотача око HDL модула је наведен.

```
class <hdl_wrapper> : public sc_core::sc_foreign_module
{
public:
    <hdl_wrapper>(sc_core::sc_module_name name) :
        sc_core::sc_foreign_module(name),
        // Member initializator list of all inputs and outputs
        ...
    {
```

```
    }

    // Inputs and outputs of HDL module
    ...

    const char* hdl_name() const;
};
```

У оквиру осталих SystemC модула, омотач класа користи се као обичан SystemC модул. Процедура за симулацију мешаног система иста је као и процедура за симулацију система у којима се користи само један језик за моделовање. Сви модели се анализирају и елаборирају у одговарајуће библиотеке. Увек постоје посебне команде за анализу HDL и SystemC модела, да би се пружила додатна флексибилност приликом развоја. Понекад је на располагању и једна команда која анализира све моделе заједно и пружа једноставност. Затим следи корак који није обавезан, а то је оптимизација модела. На крају је потребно покренути симулацију. У даљем тексту, користиће се једноставније врсте команди, када год је то могуће.

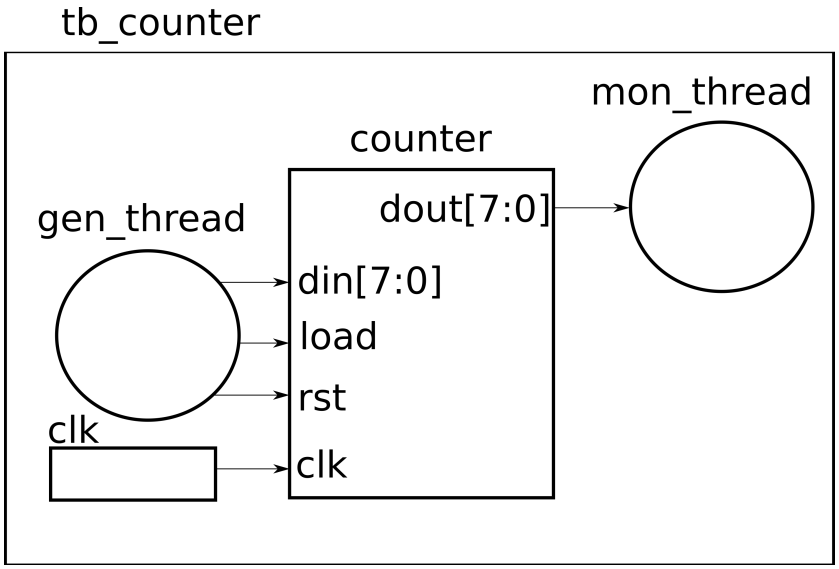
2.1.2 Пример симулације система моделованог у различитим језицима

Једноставан пример система, на коме ће бити илустровано како се симулира модел развијан у различитим језицима, састојаће се од неколико компоненти (слика 2.1). Прва компонента биће бројач `counter` развијен у неком од HDL језика. Друга компомента биће мали тестбенч, који инстацира бројач, прави стимулус и прати излазе. Инстанца `sc_clock` канала (`clk`) прави стимулус за `clk` улаз. Процес `gen_thread` прави стимулус за све остале улазе, док процес `mon_thread` посматра излазе.

Једноставан бројач може бити моделован RTL стилом моделовања (листинг 2.1). Да би се овај модел могао користити унутар SystemC модела, потребно је направити SystemC класу која је омотач око овог модула, наслеђивањем класе `sc_foreign_module` (листинг 2.2), праћењем већ описаног шаблона.

Листинг 2.1: HDL модел бројача

```
module counter (
                dout,
                clk, rst, load, din
                );
    input clk;
    input rst;
```



Слика 2.1: Систем SystemC и HDL модела

```
    input load;
    input [7:0] din;
    output [7:0] dout;

    reg [7:0]    cnt;

    always @ ( posedge clk )
      begin
          if (rst)
            cnt <= 8'd0;
          else
            if (load)
              cnt <= din;
            else
              cnt <= cnt + 1'b1;
      end

    assign dout = cnt;
endmodule
```

Листинг 2.2: Омотач класа бројача

```
#ifndef _COUNTER_HPP_
#define _COUNTER_HPP_

#include <systemc>

class counter : public sc_core::sc_foreign_module
{
public:
        counter(sc_core::sc_module_name name) :
                sc_core::sc_foreign_module(name),
                clk("clk"),
                rst("rst"),
                load("load"),
                din("din"),
                dout("dout")
        {

        }

        sc_core::sc_in< bool > clk;
        sc_core::sc_in< sc_dt::sc_logic > rst;
        sc_core::sc_in< sc_dt::sc_logic > load;
        sc_core::sc_in< sc_dt::sc_lv<8> > din;
        sc_core::sc_out< sc_dt::sc_lv<8> > dout;
```

```
        const char* hdl_name() const { return "counter"; }
};

#endif
```

Једноставан тестбенч инстанцира бројач као и сваки други SystemC модул (листинг 2.3). Променљива `dut` представља инстанцу бројача. Процес `gen_thread` прави стимулус, док процес `mon_thread` прима одзив бројача и исписује поруке. Декларисани су и одговарајући сигнали (`clk`, `rst`, `load`, `din` и `dout`), помоћу којих се шаље стимулус и прима одзив од Design Under Test (DUT) компоненте.

Комерцијални симулатори су првобитно били намењени симулацији модела развијаних у HDL језицима. У оквиру ових језика, није постојала предефинисана улазна тачка за симулацију, као што је то у SystemC језику `sc_main`. Почетни модул за симулацију се експлицитно дефинише, приликом покретања HDL симулације и обично се зове топ (од енглеске речи "top"). Комерцијални симулатори су додали могућност и за SystemC да се експлицитно дефинише топ модул. Да би модул могао да буде почетна тачка симулације, то се мора нагласити током моделовања. За ту сврху користи се макро `SC_MODULE_EXPORT`. Компоненте које су означене овим макроом, могу бити топ модули при симулацији, у оквиру комерцијалних симулатора.

Уколико је топ модул експлицитно дефинисан, SystemC симулација не сме садржати `sc_main` функцију. Приликом превођења, уколико постоји функција `sc_main`, дефинисан је макро `SC_MAIN`. Заглавље једностаног тестбенча садржи означавања класе `tb_counter` као кандидата за топ модул, уколико није дефинисана главна SystemC функција (листинг 2.3).

Листинг 2.3: Заглавље једноставног тестбенча

```
#ifndef TB_COUNTER
#define TB_COUNTER

#include <systemc>
#include "counter.hpp"

class tb_counter : public sc_core::sc_module
{
public:
        tb_counter(sc_core::sc_module_name name);
protected:
        void gen_thread();
        void mon_thread();
        counter dut;
        sc_core::sc_clock clk;

        sc_core::sc_signal< sc_dt::sc_logic > rst;
        sc_core::sc_signal< sc_dt::sc_logic > load;
        sc_core::sc_signal< sc_dt::sc_lv<8> > din;
        sc_core::sc_signal< sc_dt::sc_lv<8> > dout;
private:
};

#ifndef SC_MAIN
SC_MODULE_EXPORT(tb_counter)
#endif

#endif
```

У оквиру имплементације тестбенча, бројач се налази у листи за иницијализацију, као и сваки други SystemC модул (листинг 2.4). У оквиру конструктора, DUT је повезан са одговарајућим сигналима, који се потом користе у процесима тестбенча.

Листинг 2.4: Имплементација једноставног тестбенча

```
#include "tb_counter.hpp"
#include <string>
#include <sstream>

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

SC_HAS_PROCESS(tb_counter);

tb_counter::tb_counter(sc_module_name name) :
        sc_module(name),
        dut("dut"),
        clk("clk", 5, SC_NS)
{
        SC_THREAD(gen_thread);
        SC_METHOD(mon_thread);
        dont_initialize();
        sensitive << dout;
        dut.clk( clk.signal() );
        dut.rst( rst );
        dut.load( load );
        dut.din( din );
        dut.dout( dout );
}

void tb_counter::gen_thread()
{
        rst.write( SC_LOGIC_1 );
        load.write( SC_LOGIC_0 );
        din.write( 7 );
        wait(100, SC_NS);
        rst.write( SC_LOGIC_0 );
        wait(500, SC_NS);
        load.write( SC_LOGIC_1);
        wait(100, SC_NS);
        load.write( SC_LOGIC_0);
        wait(1000, SC_NS);
        sc_stop();
}

void tb_counter::mon_thread()
{
        ostringstream ss;
        ss << "@" << sc_time_stamp();
        ss << " dout = " << dout.read();
        ss << " ("<< static_cast< sc_uint<8> > (dout.read()) << ")";
        SC_REPORT_INFO(name(), ss.str().c_str());
}
```

Главни програм за овај модел, инстанцира тестбенч и покреће симулацију (листинг 2.5). Додатно, дефиниса је и вредност `SC_MAIN`, која ће означити да постоји главна функција, уколико се она буде користила приликом покретања симулације.

Листинг 2.5: Главни програм

```
#define SC_MAIN
#include <systemc>
#include "tb_counter.hpp"

using namespace sc_core;

int sc_main(int argc, char* argv[])
{
  tb_counter uut("uut");
```

```
  sc_start();

  return 0;
}
```

Након што су сви модели припремљени, симулација се у оквиру програмског пакета Xcelium може покренути на два начина. Уколико се користи главна SystemC функција за симулацију, позива се наредба:

```
xmsc_run -sc_main -gui sc_main.cpp tb_counter.cpp counter.v
```

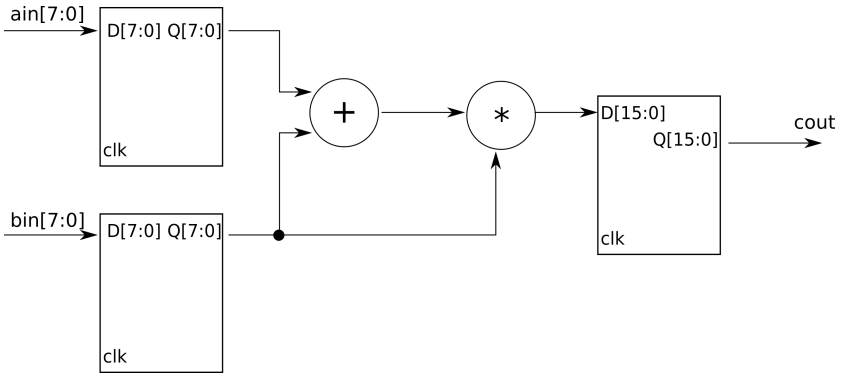
Параметар `-sc_main` означава да се користи главна функција. Параметар `-gui` омогућава да се отвори GUI након анализе, елаборације и покретања симулације. Уколико се не користи главна функција, тада се симулација покреће командом:

```
xmsc_run -gui tb_counter.cpp counter.v -top tb_counter
```

Вредност улазног аргумента прослеђеног са `-top`, означава који модул је топ за покренуту симулацију и то је `tb_counter`. У овом случају се аргумент `-sc_main` не користи.

2.1.3 Вежбе

1. У неком од HDL језика, моделовати систем који садржи регистре, сабираче и множаче (слика 2.2). Уградити овај модел, као DUT, у тестбенч моделован у SystemC језику. Тестбенч треба да је самопроверавајући.



Слика 2.2: Дигитални систем са регистрима, сабирачем и множачем

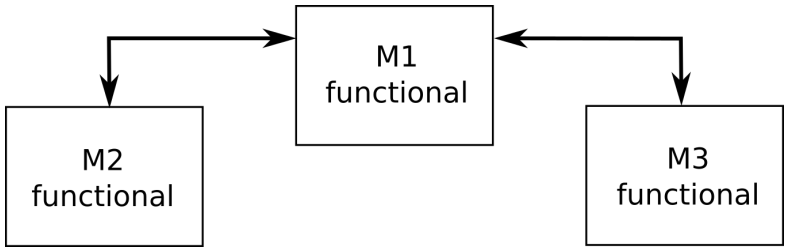
2.2 Профињавање модела

Пројектовање засновано на профињавању модела је начин пројектовања система код кога се постепено прелази од описа система на високом нивоу абстракције ка опису система на ниском нивоу абстракције. Почетни модели система су абстрактни, секвенцијални и време се не исказује. Делови система комуницирају међусобно позивима функција. Стога се ови абстрактни модели често називају функционални.

Током пројектовања део по део система се мења моделима који садрже већу количину детаља. Модели постају у већој мери конкретни, конкурентни и време се укључује у опис система. У току профињавања дефинишу се компоненете које ће постојати у коначној имплементацији као и интерфејси којима ће компоненете комуницирати. Као коначни резултат овог процеса добија се опис система, који је могуће имплементирати у одговарајућој технологији.

У овом процесу потребно је моделовати и софтверске као и хардверске делове система. У SystemC језику се ово може решити на добар начин и то је главни разлог зашто се овај језик користи за пројектовање на системском нивоу. Механизам хијерархијских канала игра битну улогу у овом процесу. Размотримо овај начин пројектовања, прво на једном абстрактном примеру а потом на конкретном примеру.

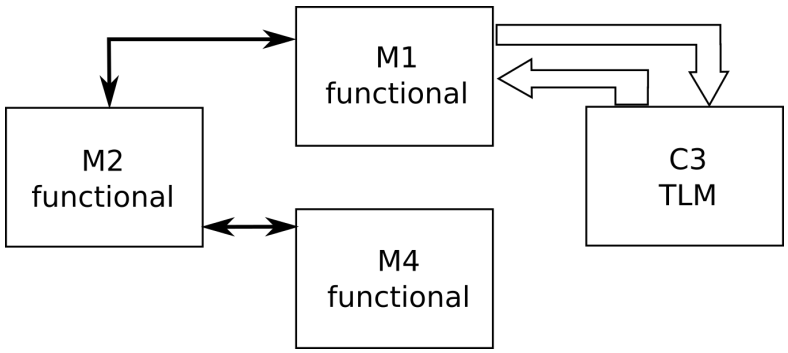
За абстрактни пример узмимо систем који се састоји од три компоненте (слика 2.3). У самом почетку пројектовања све три компоненте (`M1`, `M2` и `M3`) биће описане на функционалном нивоу абстракције. Комуникација ће бити помоћу функција (задебљане стрлице). У разматраном систему комуникација постоји између компоненти `M1` и `M2`, као и `M1` и `M3`. Ови модели могу бити представљеени обичним C++ функцијама или методама класа. Цео систем је тестиран неким улазним подацима и даје коректне излазне податке.



Слика 2.3: Почетни функционални модел

Претпоставимо да је компонента `M2` сувише комплексна за даљу имплементацију. У наредној фази (слика 2.4) део функционалности

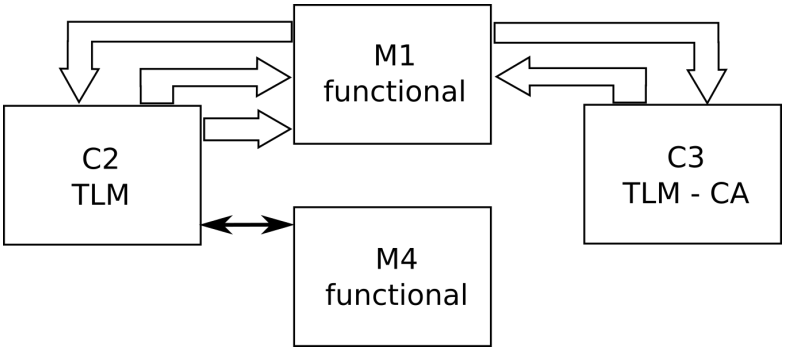
компоненте M2 пребацује се у нову компоненту M4, да би се систем могао лакше имплементирати. Даље, предпоставимо да је компонента M3 довољно једноставна да може да се имплементира и да је јасно какве су трансакције потребне између делова система M1 и M3. Компонента M3 може се моделовати као SystemC канал (C3) са јасно дефинисаним интерфејсима (стреле). Рецимо да се приликом моделовања компоненте није узело време у обзир, тако да је ова компонента моделована на Transaction Level Modeling (TLM) нивоу абстракције. У овај фази се прешло са описа система у језику C++ на опис система у SystemC језику. Овај профињени систем би требао да се тестира истим улазним подацима као и почетни систем и требао би да даје идентичне излазне податке. Ово је битна добит овог начина пројектовања. Цео систем се може континуирано верификовати, без пројектовања додатног верификационог окружења.



Слика 2.4: Профињење фаза 1

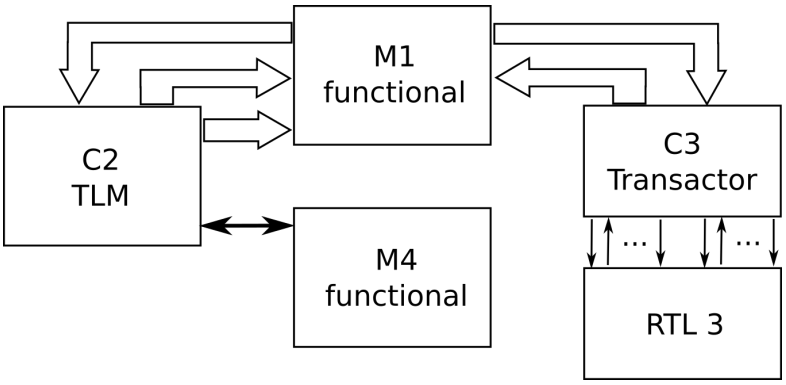
Предпоставимо да је компонента M2 довољно једноставна да може даље да се профини. Нека је у овом тренутку јасно које трансакције треба да се размењују између компоненти M1 и M2. Тада се компонента M2 може моделовати као хејерахијски канал са јадно дефинисаним интерфејсима (слика 2.5). У модел канала C3 додају се и временски детаљи и ова компонента је још један ниво абстракције ниже, на TLM новоу али са апроксимацијом времена. Цео систем се и даље може симулирати у симулатору који подржава само SystemC и може се верификовати као и претходна два модела целог система.

Као последњу фазу овог абстрактног примера, узећемо да је канал C3 подељен на трансактор и RTL модул (слика 2.6). Трансактор преузима абстрактне трансакције између модула M1 и канала C3 и претвара их на промене сигнала између трансактора и RTL 3 модула (неподебљане стрлице на слици). Ова промена нема утицаја на остатак система. RTL модул је најчешче имплементиран у неком од HDL



Слика 2.5: Профињење фаза 2

језика. Почетни модул `M3` профињен је са довољно детаља да може да се имплементира. Да би се овај систем могао симулирати потребан је симулатор који подржава мешану симулацију SystemC модела и HDL модела. Комерцијални симулатори имају подршку за овакав тип симулација. Цео систем се и даље верификује на потпуно исти начин.



Слика 2.6: Профињење фаза 3

Овај абстрактни пример је илустровао како се пројектује на бази профињавања модела. Из фазе у фазу се методички додају додатни детаљи у сваки део система све до тренутка када су сви делови система могући за имплементацију у неком од алата. Што је веома битно, у току целог поступка, може се користити идентично верификационо окружење. На тај начин се може потврдити да прилико спуштања компоненти на нижи ниво абстракције није дошло до грешака.

Као конкретан пример овог поступка пројектовања, биће завршен пример са системом за сортирање бројева. Приликом развоја тог система, већ је била примењивана метода профињавања модела. Остао је још корак, којим се модел спушта на ниво абстракције, са ког је

могућа имеплемнтација. Да би се пример могао симулирати, мораће се спровести процедура са симулацију модела развијаних у различитим језицима.

2.2.1 Пример профињавања модела

Систем за сортирање је профињен до TLM нивоа са апроксимирањем времена (слика 1.39). Да би акцелератор за сортирање могао имплементирати остало је још да се модул са именом `core` који је имплементиран модулом `sort_acc_ca_calculate` спусти на RTL ниво абстракције. То се може урадити ручно коришћењем RTL методологије, почевши од алгоритма имплементираног у процесу `sort` модула `sort_acc_ca_calculate`. Други начин је да се HLS методологијом, аутоматски на основу описа алгоритма добије RTL имплементација. Како год, нека смо добили RTL имплементацију неким од поступака и нека сада имамо RTL модел овог акцелратора имплементиран у HDL језику. Нека тај модел комуницира са остатком система преко сигнала: `ap_clk`, `in_r_TREADY`, `in_r_TDATA`, `in_r_TLAST`, `in_r_TVALID`, `out_r_TREADY`, `out_r_TDATA`, `out_r_TLAST`, `out_r_TVALID`, `ap_rst_n`,

Сви примери овог система до сада, могли су да раде и на референтном OSCI симулатору. Овај пример садржи мешавину SystemC модела и HDL модела, па је потребан симулатор који има подршку за мешовиту симулацију. Додатно, потрено је направити класу, омотач, за HDL модел (листинг 2.6).

Листинг 2.6: Класа за укључивање HDL модела унутар SystemC модела

```
#ifndef _SORT_ACC_CA_CALCULATE_WRAP_
#define _SORT_ACC_CA_CALCULATE_WRAP_

#include "systemc.h"

class sort_acc_ca_calculate : public sc_foreign_module
{
public:
    sc_in<bool> ap_clk;
    sc_in<bool> ap_rst_n;
    sc_in<sc_lv<16> > in_r_TDATA;
    sc_in<bool> in_r_TVALID;
    sc_out<bool> in_r_TREADY;
    sc_in<bool> in_r_TLAST;
    sc_out<sc_lv<16> > out_r_TDATA;
    sc_out<bool> out_r_TVALID;
    sc_in<bool> out_r_TREADY;
    sc_out<bool> out_r_TLAST;


    sort_acc_ca_calculate(sc_module_name nm, const char* hdl_name,
        int num_generics, const char** generic_list)
     : sc_foreign_module(nm),
       ap_clk("ap_clk"),
       ap_rst_n("ap_rst_n"),
       in_r_TDATA("in_r_TDATA"),
       in_r_TVALID("in_r_TVALID"),
       in_r_TREADY("in_r_TREADY"),
       in_r_TLAST("in_r_TLAST"),
       out_r_TDATA("out_r_TDATA"),
       out_r_TVALID("out_r_TVALID"),
```

```
        out_r_TREADY("out_r_TREADY"),
        out_r_TLAST("out_r_TLAST")
    {
        elaborate_foreign_module(hdl_name, num_generics, generic_list);
    }
    ~sort_acc_ca_calculate()
    {}

};

#endif
```

Након ове припреме, HDL модел се користи идентично SystemC модулима. У разматраном примеру система за сортирање, у класи која моделује акцелератор апроксимирајући време (листинг 1.45), илустрован је начин коришћења HDL модела. Део модела који подразумева да је дефинисана предпроцесорска директива COSIM служи да инстанцира HDL компоненту. Прво, у модел се убацује заглавље у коме је класа која наслеђује `sc_foreign_module`, уместо оригиналног заглавља у коме је SystemC модел са апроксимативним временом. Друго, у конструктору се инстацира, та нова класа уместо оригиналне. Имена класа су идентична, а разликују се имена сигнала, као и параметри конструктора. Остатак модела остаје потпуно исти.

Симулација потврђује да ли је RTL модел добро имплементиран. Још једном се наглашава да се остатак система не мења, већ да је једино акцелератор за сортирање профињен на ниво абстракције са ког може да се имплементира. Овај модел даље може да се користи у алатима за синтезу са RTL нивоа абстракције, као што је Vivado, на пример.

2.2.2 Вежбе

1. Пројектовати RTL модел неког филтра 5. реда. Са већ развијеним тестбенчом за хијерархијске канале, верификовати функционалност RTL имплементације.

2. Пројектовати RTL модел генератора Фибоначијевих бројава. Са већ развијеним тестбенчом за хијерархијске канале, верификовати функционалност RTL имплементације.