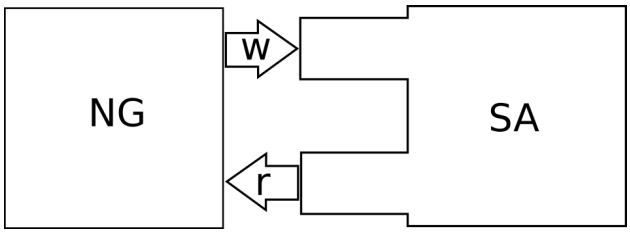
1.10 Хијерархијски канали

Хијерархиски канали су основа за системско моделовање у језику SystemC. Канал је нешто што имплементира један или више SystemC интерфејса. Интерфејс је скуп декларација метода помоћу којих се може приступити неком каналу. Раздвајањем декларације интерфејса од имплементације његових метода омогућено је једноставно мењање нивоа абстракције приликом моделовања система. Додатно, коришћење портова олакшава повезивање модула са различитим каналима.

Под хијерархијским каналом у SystemC језику може се подразумевати било који дигитални систем. Дигитални систем, у општем случају, прима информације, обрађује их, и потом шаље резултате обраде. Овај опис дигиталног система одговара опису хијерархијског канала.

1.10.1 Пример моделовања помоћу хијерархијских канала

Главну добит једноставног прелажења са једног нивоа абстракције на други, коришћењем хијерархијских канала биће илустровано на примеру (слика 1.31). Пример се састоји од две компоненете. Једна компонента је модул који преко интерфејса за писање шаље низ бројева другој компоненти. Ова компонента је модул и означена је са NG (енг. Number generator). Друга компонента је дигитални систем који сортира низ бројева. Ова компонента је хијерархијски канал и означена је са SA (енг. Sort Accelerator). Та компонента је канал који бројеве добија преко једног интерфејса за писање и резултате шаље преко интерфејса за читање. Стрелица са ознаком `w` је порт за писање компоненте NG, а стрелиза `r` је порт за читање. Канал SA имплементираће, на различите начине, операцију сортирања и биће закачен на ова два порта компоненте NG.



Слика 1.31: Систем са хијерархијским каналом

Правила комуникације за овај систем су дефинисана преко два интерфејса: интерфејс за писање `sort_write_if` и интерфејс за

читање `sort_read_if` листинг 1.30. Ови интерфејси наслеђују класу `sc_interface`, што значи да су намењени дефинисању правила комуникације. Зашто су наслеђени са кључном речју `virtual` биће објашњено у једном од наредних параграфа.

Листинг 1.30: Интерфејси за читање и писање

```
#ifndef _SORT_IFS_HPP_
#define _SORT_IFS_HPP_

#include <systemc>
#include <vector>

class sort_write_if : virtual public sc_core::sc_interface
{
public:
        virtual void write(const std::vector<int>& data) = 0;
};

class sort_read_if : virtual public sc_core::sc_interface
{
public:
        virtual void read(std::vector<int>& data) = 0;
};

#endif
```

Интерфејс за писање садржи једну методу названу `write`. Ова метода је чиста виртуелна метода. То значи да класе које наследе овај интерфејс морају да имплементирају ову методу, уколико се користе у моделу. Ова метода као улазни параметар прима низ целих бројева. Овај низ је означен модификатором `const`, што значи да низ неће бити мењан унутар методе. На овај начин, интерфејс за писање дефинисао је дозвољена правила комуникације кроз било који канал који имплементира овај интерфејс. Та правила захтевају да се за комуникацију користи медода `write` у коју се проследи низ целобројних елеменаза који се шаљу. При томе, метода гарантује да се улазни низ неће мењати.

Слично је и са интерфејсом за читање. И ова класа садржи једну методу која је чиста виртуелна и зове се `read`. Ова метода као параметар прима низ целих бројева, али он није означен кључном речју const. То значи да се овај низ може мењати у имплементацији ове методе. То значи да интерфејс за читање дефинише да сваки канал који имплементира овај интерфејс, за комумуникацију могу да користе само методу `read`. Имплементација ове метода може прочитане вредности да поређа у низ који се прослеђује као параметар.

Уз помоћ ова два интерфејса дефинисана су правила комуникације у систему. Једна компонента попуниће низ целих бројева неким вредностима. Те вредности послаће коришћењем интерфејса за писање некој другој компоненти. Резултате сортирања примиће преко интерфејса за читање. Одговорности компоненти су јасно раздвојене уз помоћ интерфејса.

Листинг 1.31: Декларација класе за слање несортираних бројева

```
#ifndef _SORT_GEN_CHECK_HPP_
#define _SORT_GEN_CHECK_HPP_

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

SC_MODULE(sort_gen_check)
{
public:
        SC_HAS_PROCESS(sort_gen_check);

        sort_gen_check(sc_core::sc_module_name);

        sc_core::sc_port< sort_write_if > wr_port;
        sc_core::sc_port< sort_read_if > rd_port;
protected:
        void test();
        std::vector<int> gen_nums;
};

#endif
```

Компонента која шаље бројеве (`sort_gen_check`) који нису сортирани не треба да зна ништа о начину на који ће бројеви бити сортирани (листинг 1.31). Преко портова је обезбеђена комуникација са остатком система (`wr_port`, `rd_port`). Уметсо портова могли су бити коришћени и показивачи на интерфејсе (`sort_write_if*` и `sort_read_if*`) али се у том случју не би могло користити именовано повезивање портова на конкретан канал.

Листинг 1.32: Имплементација класе за слање несортираних бројева

```
#include "sort_gen_check.hpp"
#include <algorithm>

using namespace std;
using namespace sc_core;

sort_gen_check::sort_gen_check(sc_module_name name) :
        sc_module(name)
{
  cout << "Constructed\n";
  SC_THREAD(test);
}

void sort_gen_check::test()
{
  for(int testnum = 0; testnum != 10; testnum++)
        {
          //    Generate lengths from 10 to 100
          int len = 10 + rand() % 90;
          vector<int> tran;

          cout << "Generate transaction of " << len << " integers.\n";

          for (int i=0; i != len; ++i)
                {
                  int num = rand() % 1000;
                  tran.push_back(num);
                }

          wr_port->write(tran);
          cout << "Transaction sent.\n";


          sort(tran.begin(), tran.end());

          vector<int> res;
          rd_port->read(res);
          cout << "Transaction received.\n";

          if(tran.size() != res.size())
                {
                  cout << "Test failed.\n";
                  cout << "Expected size " << tran.size() << endl;
```

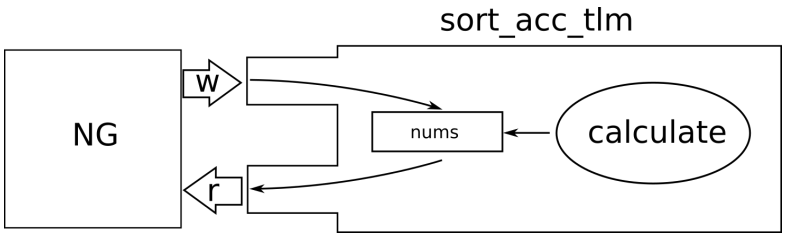
```
                cout << "Received size " << res.size() << endl;
                return;
              }

        for (size_t i = 0; i != tran.size(); ++i)
              {
                if(tran[i] != res[i])
                      {
                        cout << "Test " << testnum << " failed.\n";
                        cout << "Elemnt with index " << i << endl;
                        cout << "Expected value " << tran[i] << endl;
                        cout << "Received value " << res[i] << endl;
                        return;
                      }
              }

        cout << "Test " << testnum << " passed.\n";
      }
}
```

Ова компонента садржи и један процес (`test`) у коме се праве трансакције. Трансакције су помоћни низ (`tran`, листинг 1.32) који се потом прослеђује у порт за слање. Низ се попуњава на случајан начин бројевима од 1 до 100. Исти процес потом сортира низ који је послао. За сортирање се користи функција `sort`, која се укључује заглављем `algorithm`. Након сортирања, процес прима резултујући низ који је сортирао остатак система (`res`) преко порта за читање. На крају, овај процес проверава да ли се очекивани резултат соритрања поклапа са примљеним резултатом. Исписују се и одговарајуће поруке.

Описана компонента (`sort_gen_check`) представља мали тестбенч са остатак система. Она ствара трансакције, предвиђа резултат, прима трансакције и проверава очекиване и добијене вредности. Битно је нагласити да је цела имплементација урађена без познавања остатка система, пошто је урађено раздвајање одговорности. Додатно све ради преко хијерархијских интерфејса на нивоу трансакција. Процес компоненте (`test`) не троши време.

На ову компоненту се сада могу повезати различите имплементације канала који реализују интерфејсе за читање и писање. Прво ће бити разматран случај канала чија је имплементација у потпуности на нивоу трансакција и чије методе не троше симулационо време. Овај канал биће имплементиран класаом која се зове `sort_acc_tlm` (Sort Accelerator Transaction Level Model, слика 1.32). Унутар ове класе постојеће један процес (`calculate`) који ће сортирати примљене бројеве, при чему неће трошити симулационо време.

Декларација канала који имплементира канал за сортирање бројева користи механизам вишеструког наслеђивања (листинг 1.33). Наслеђивањем класе `sc_channel` означава се да је класа хијерархијски канал. Наслеђивањем интерфејса за читање и писање (`sort_write_if` и `sort_read_if`) одређује се шта ће све канал да имплементира. Ово је и добро место да се објасни зашто су интерфејси за писање и слање виртуелно наследили класу `sc_interface` (1.30).

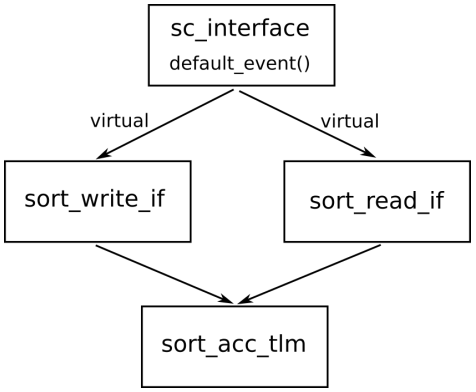


Слика 1.32: Систем са каналом који не троши симулационо време

Листинг 1.33: Декларација канала за сортирање на нивоу трансакција

```
#ifndef _SORT_ACC_TLM_HPP_
#define _SORT_ACC_TLM_HPP_

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

class sort_acc_tlm :
        public sc_core::sc_channel,
        public sort_write_if,
        public sort_read_if
{
public:
        SC_HAS_PROCESS(sort_acc_tlm);

        sort_acc_tlm(sc_core::sc_module_name);
        void write(const std::vector<int>& data);
        void read(std::vector<int>& data);
        void calculate();
protected:
        std::vector<int> nums;
};

#endif
```

Виртуелно наслеђивање је врста C++ вишеструког наслеђивања која осигурава да се само једна метода преузима од базне класе која је бар два нивоа хијерархије изнад. У примеру, `sort_write_if` и `sort_read_if` наслеђују `sc_interface` (слика 1.33). Класа `sc_interface` садржи методу `default_event`. Класа `sort_acc_tlm` наслеђује и `sort_write_if` и `sort_read_if`.



Слика 1.33: Виртуелно наслеђивање

Уколико се не би користило виртуелно наслеђивање, тада би класа `sort_acc_tlm` садржала две методе `default_event`, једну коју би преузела од `sort_read_if` а другу од `sort_write_if`. Те две метода би се морале позивати независно коришћењем оператора за разрешавање опсега. Када се користи виртуелно наслеђивање, тада класа `sort_acc_tlm` садржи само једну методу `default_event`, наслеђену од класе `sc_interface`. Класа `sort_acc_tlm` треба да садржи само по једну методу наслеђену од класе `sc_interface` и стога је потребно користити виртуелно наслеђивање. Генерално, када год је класа намењена за вишеструко наслеђивање, потребно је користити виртуелно наслеђивање.

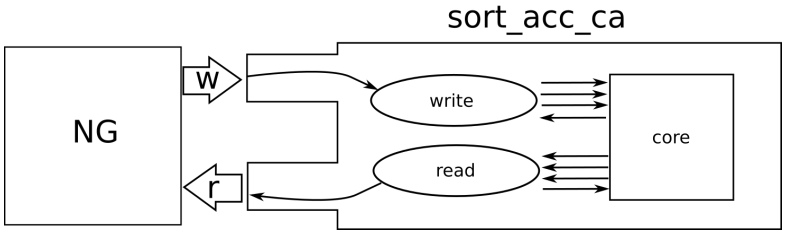
Имплементација класе `sort_acc_tlm` је написана на нивоу абстракције трансакција (листинг 1.34). Методом наслеђеном од интерфејса за писање (`write` метода), улазни низ се пребацује у интерни низ (`nums`). Након тога се позива метода која сортира интерни низ (`calculate`). Ова метода не троши симулационо време. Методом наслеђеном од интерфејса за читање, интерни низ се пребацује у низ прослеђен као параметар (`read` метода).

Листинг 1.34: Модел акцелератора за сортирање који не троши симулационо време

```
#include "sort_acc_tlm.hpp"

using namespace std;
using namespace sc_core;

sort_acc_tlm::sort_acc_tlm(sc_module_name name) : sc_channel(name)
{
        cout << "Constructed.\n";
}

void sort_acc_tlm::write(const std::vector<int>& data)
{
        nums.clear();
        for (size_t i = 0; i != data.size(); ++i)
                nums.push_back(data[i]);
        cout << "Date of size " << data.size() << " writen to accelerator.\n";

        calculate();
}

void sort_acc_tlm::calculate()
{
        for (size_t i=0; i != nums.size(); ++i)
                for(size_t j=i+1; j != nums.size(); j++)
                        if (nums[i] > nums[j])
                        {
                                int tmp = nums[i];
                                nums[i]= nums[j];
                                nums[j] = tmp;
                        }
}

void sort_acc_tlm::read(std::vector<int>& data)
{
        data.clear();
        for (size_t i = 0; i != nums.size(); ++i)
                data.push_back(nums[i]);
        cout << "Date of size " << data.size() << " read from accelerator.\n";
}
```

Сортирање је урађено без коришћења готове функције, за разлику од имплементације сортирања у модулу који генерише и проверава трансакције. На овај начин су референтна имплементација и имплементација унутар класе `sort_acc_tlm` урађене на потпуно другачији начин, што је веома важно. Имплементација унутар класе `sort_acc_tlm` је на вишем нивоу абстракције од RTL нивоа, али је на нижем од референтне имплементације.

Размотримо сада другачију имплементацију овог канала. Имплементација ће бити на нижем нивоу абстракције од претходно разматране. Користиће се и сигнали за комуникацију и пренос трансакција. Методе ове имплементације трошиће време. Апроксимираће број циклуса неопходних за прорачун који би био потребан да се направи овакав акцелератор RTL или HLS методологијом. Класа `sort_acc_ca` (Sort Accelerator Cycle Accurate, слика 1.34) имплементираће овакав канал који ће садржати и додатну компоненту (`core`) са којом комуницира сигналима.



Слика 1.34: Систем са каналом који троши симулационо време

Јавни интерфејс канала за сортирање апроксимативног у циклусима (листинг 1.35) је идентичан интерфејсу канала који не троши време. Користи се исти механизам вишеструког наслеђивања. У недоступном делу интерфејса налазе се знатне разлике. Декларисан је велик број сигнала (`in_tready`, `in_tdata`...). Унутар класе се налази и генератор синхронизационог сигнала (`clk`). Декларисана је и компонента помоћу које ће се апроксимирати број потребних циклуса за прорачун (`sort_acc_ca_calculate`).

Листинг 1.35: Декларација канала за сортирање апроксимативног у циклусима

```
// Cycle accurate model of sort accelerator

#ifndef _SORT_ACC_CA_HPP_
#define _SORT_ACC_CA_HPP_

#include <systemc.h>
#include "sort_ifs.hpp"

class sort_acc_ca_calculate;
```

```
class sort_acc_ca :
        public sc_core::sc_channel,
        public sort_write_if,
        public sort_read_if
{
public:
        SC_HAS_PROCESS(sort_acc_ca);

        sort_acc_ca(sc_core::sc_module_name name);

        void write(const std::vector<int>& data);
        void read(std::vector<int>& data);

        sc_core::sc_clock clk;
        sc_core::sc_signal< bool > rst;

        sc_core::sc_signal< sc_dt::sc_logic > in_tready;
        sc_core::sc_signal< sc_dt::sc_lv<16> > in_tdata;
        sc_core::sc_signal< sc_dt::sc_logic > in_tlast;
        sc_core::sc_signal< sc_dt::sc_logic > in_tvalid;

        sc_core::sc_signal< sc_dt::sc_logic > out_tready;
        sc_core::sc_signal< sc_dt::sc_lv<16> > out_tdata;
        sc_core::sc_signal< sc_dt::sc_logic > out_tlast;
        sc_core::sc_signal< sc_dt::sc_logic > out_tvalid;


        friend void sc_trace(sc_core::sc_trace_file*, const sort_acc_ca&, const std::string& name);
protected:
        sort_acc_ca_calculate* core;
};

#endif
```

Конструктор класе инстанцира додатну компоненту (листинг 1.36) и повезује је са својим унутрашњим сигналима. За тренутно разматрање узећемо само део модела који подразумева да није дефинисана предпроцесорска директива `COSIM`. Метода наслеђена од интерфејса за писање (`write`) преводи улазну трансакцију на промене у сигналима неопходне да би унутрашња компонента могла да уради прорачун. Понекад се оваква метода назива и трансактор, а понекад се имплементира и као посебан модул. Слично је и са методом наслеђеном од интерфејса за читање, само се претварање ради у супротном смеру: промене на сигналима се претварају у излазну трансакцију која се прослеђује остатку систеа преко излазног низа (`data`).

Листинг 1.36: Модел акцелератора за сортирање апроксимативног у циклусима

```
#include "sort_acc_ca.hpp"
#ifdef COSIM
#include "sort_acc_ca_calculate_rtl.hpp"
#else
#include "sort_acc_ca_calculate.hpp"
#endif

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

sort_acc_ca::sort_acc_ca(sc_module_name name) :
        sc_module(name),
        clk("m_clk", 100, SC_NS, 0.4, 5, SC_NS, true),
        rst("rst"),
        in_tready("in_tready"),
        in_tdata("in_tdata"),
        in_tlast("in_tlast"),
        in_tvalid("in_tvalid"),
        out_tready("out_tready"),
```

```
        out_tdata("out_tdata"),
        out_tlast("out_tlast"),
        out_tvalid("out_tvalid")
{
#ifndef COSIM
        core = new sort_acc_ca_calculate("core");

        core->clk(clk);
        core->in_tready(in_tready);
        core->in_tdata(in_tdata);
        core->in_tlast(in_tlast);
        core->in_tvalid(in_tvalid);
        core->out_tready(out_tready);
        core->out_tdata(out_tdata);
        core->out_tlast(out_tlast);
        core->out_tvalid(out_tvalid);
        core->rst(rst);
#else
        core = new sort_acc_ca_calculate("core", "sort_acc_ca_calculate");

        core->ap_clk(clk);
        core->in_r_TREADY(in_tready);
        core->in_r_TDATA(in_tdata);
        core->in_r_TLAST(in_tlast);
        core->in_r_TVALID(in_tvalid);
        core->out_r_TREADY(out_tready);
        core->out_r_TDATA(out_tdata);
        core->out_r_TLAST(out_tlast);
        core->out_r_TVALID(out_tvalid);
        core->ap_rst_n(rst);
#endif
}


void sort_acc_ca::write(const std::vector<int>& data)
{
        rst.write(false);
        out_tready.write(SC_LOGIC_0);
        in_tvalid.write(SC_LOGIC_0);
        in_tlast.write(SC_LOGIC_0);

        for (int i = 0; i != 10; ++i)
                wait(clk.negedge_event());

        for (size_t i = 0; i != data.size(); ++i)
        {
                sc_lv<16> v(data[i]);
                in_tdata = v;
                in_tvalid = SC_LOGIC_1;
                if (i == (data.size()-1))
                        in_tlast = SC_LOGIC_1;
                else
                        in_tlast = SC_LOGIC_0;
                while(true)
                {
                        wait(clk.negedge_event());
                        if (in_tready == SC_LOGIC_1) break;
                }
        }
        cout << "Transaction translated.\n";
        in_tvalid = SC_LOGIC_0;
        in_tlast = SC_LOGIC_0;
}

void sort_acc_ca::read(std::vector<int>& data)
{
        rst.write(false);
        data.clear();
        wait(clk.negedge_event());
        out_tready = SC_LOGIC_1;

        while(1)
        {
                wait(clk.negedge_event());
                if (out_tvalid == SC_LOGIC_1)
                {
                        sc_lv<16> v = out_tdata;
                        data.push_back(v.to_int());
                        if (out_tlast == SC_LOGIC_1)
                                break;
                }
        }
```

```
        wait(clk.negedge_event());
        out_tready = SC_LOGIC_0;
}

void sc_trace(sc_core::sc_trace_file* tf, const sort_acc_ca& obj, const std::string& name)
{
        sc_trace(tf, obj.clk, name + ".clk");
        sc_trace(tf, obj.in_tready, name + ".in_tready");
        sc_trace(tf, obj.in_tdata, name + ".in_tdata");
        sc_trace(tf, obj.in_tlast, name + ".in_tlast");
        sc_trace(tf, obj.in_tvalid, name + ".in_tvalid");
        sc_trace(tf, obj.out_tready, name + ".out_tready");
        sc_trace(tf, obj.out_tdata, name + ".out_tdata");
        sc_trace(tf, obj.out_tlast, name + ".out_tlast");
        sc_trace(tf, obj.out_tvalid, name + ".out_tvalid");
}
```

За имплементацију ових метода користе се методе сигнала. Све је синхронизовано помоћу унутрашњег генератора синхронизационог сигнала. Улазни цели бројеви се конвертују у 16-битне логичке вредности, апроксимирајући на тај начин прецизност коначне хардверске имплементације.

Овај канал садржи и `friend` C++ слободну функцију, `sc_trace`, у својој декларацији. Оваква функција са тачно таквим наведеним параметрима омогућује новим модулима и каналима да избаце своје вредности у Value Change Dump (VCD) датотеке, које се после могу посматрати у симулаторима. Ово је исти механизам као и код проширивања могућности приказа ostream класе имплементирањем оператора «. Дефиниција ове слободне функције се налази заједно са дефиницијама осталих метода овог канала.

Да би завршили разматрање апроксимативног система, потребно је погледати и имплементацију унутрашње компоненте која сортира апроксимирајући потребни број циклуса (`sort_acc_ca_calculate`, листинг 1.37). Интерфејс ове компоненте ка остатку система су само сигнали што значи да је компонента приближна коначној имплементацији овог акцелератора. Под коначном имплементацијом најчешће се подразумева RTL модел.

Листинг 1.37: Декларација компоненте чији интерфејс су само сигнали

```
#ifndef _SORT_ACC_CA_CALCULATE_HPP_
#define _SORT_ACC_CA_CALCULATE_HPP_

#include <systemc>

SC_MODULE(sort_acc_ca_calculate)
{
public:
        SC_HAS_PROCESS(sort_acc_ca_calculate);

        sort_acc_ca_calculate(sc_core::sc_module_name);

        sc_core::sc_in< bool > clk;
        sc_core::sc_in< bool > rst;

        sc_core::sc_out< sc_dt::sc_logic > in_tready;
        sc_core::sc_in< sc_dt::sc_lv<16> > in_tdata;
        sc_core::sc_in< sc_dt::sc_logic > in_tlast;
        sc_core::sc_in< sc_dt::sc_logic > in_tvalid;

        sc_core::sc_in< sc_dt::sc_logic > out_tready;
```

```
        sc_core::sc_out< sc_dt::sc_lv<16> > out_tdata;
        sc_core::sc_out< sc_dt::sc_logic > out_tlast;
        sc_core::sc_out< sc_dt::sc_logic > out_tvalid;

protected:
        void sort();
        // void wr_data();
        // void rd_data();

        int arsize;
        typedef sc_dt::sc_int<16> num_t;
        num_t nums[10000];
};

#endif
```

Компонента `sort_acc_ca_calculate` имплементирана је као модул. Интерфејс ка остатку система је искључино преко улазних и излазних портова за сигнале. Стога, интерфејс ове компонента је врло сличан интерфејсима RTL компоненти писаним у HDL језицима. Компонента сву своју функционалност реализује једним процесом (`sort`) и садржи један бројач (`arsize`) и низ (`nums`) у који смешта бројеве за сортирање.

Листинг 1.38: Имплементација компоненте чији интерфејс су само сигнали

```
#include "sort_acc_ca_calculate.hpp"

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

sort_acc_ca_calculate::sort_acc_ca_calculate(sc_module_name name) :
        sc_module(name)
{
        SC_CTHREAD(sort, clk.pos());
        reset_signal_is(rst,true);
}


void sort_acc_ca_calculate::sort()
{
        while(1)
        {
                cout << "Starting...\n";
                arsize = 0;
                out_tvalid.write(SC_LOGIC_0);
                out_tlast.write(SC_LOGIC_0);
                in_tready.write(SC_LOGIC_1);
                wait();

                while(1)
                {
                        wait();
                        //in_tready.write(SC_LOGIC_1);
                        if (in_tvalid.read() == SC_LOGIC_1)
                        {
                                nums[arsize++] = in_tdata.read();
                                if(in_tlast.read() == SC_LOGIC_1)
                                {
                                        break;
                                }
                        }
                }

                wait();
                cout << "Received transaction of size " << arsize << endl;
                in_tready.write(SC_LOGIC_0);

                for(int i=0; i != arsize; ++i)
                        for(int j=i; j != arsize; ++j)
                        {
                                wait();
                                sc_int<16> v1 = nums[i];
                                wait();
```

```
                        sc_int<16> v2 = nums[j];
                        wait();
                        if(v1 > v2)
                        {
                                nums[i] = v2;
                                wait();
                                nums[j] = v1;
                        }
                }

        cout << "Cycle accurate core done sorting.\n";

        int i = 0;
        sc_int<16> buf = nums[0];
        wait();
        sc_int<16> buf1 = nums[1];
        wait();

        while(i != arsize)
        {

                wait();
                out_tvalid.write(SC_LOGIC_1);
                out_tdata.write(buf);
                if (i == (arsize-1))
                        out_tlast.write(SC_LOGIC_1);
                else
                        out_tlast.write(SC_LOGIC_0);
                if (out_tready.read() == SC_LOGIC_1)
                {
                        i++;
                        buf = buf1;
                        buf1 = nums[i+1];
                }
        }
        cout << "Transaction sent from cycle accurate core.\n";
        wait();
    }
}
```

Процес `sort` је имплементиран као `SC_CTHREAD` осетљив на улазни сигнал `clk`. Овај провес садржи три главна дела. У првом делу се примају подаци у унутрашни низ, помоћу сигнала који симулирају улазни интерфејс. У другом делу се сортирају подаци. У трећем делу се сортирани подаци шаљу остатку система, помоћу сигнала који симулирају излазни интерфејс. Имплементација је прожета позивима `wait` метода. На овај начин овај модел апроксимира потршњу времена коначне имплементације акцелератора за сортирање.

1.10.2 Закључак

Претходни пример је илустровао како помоћу хијерархијских канала може да се имплементација једног дела система замени другачијом имплементацијом истог тог дела система. При томе, што је најважније, остали делови система не морају уопште да се мењају. Овај механизам омогућава да се реализује методологија базирана на постепеном профињавању модела. О овој методологији биће речи у наредним поглављима.

1.10.3 Вежбе

1. Моделовати неки IIR филтар 5 реда као хијерархијски канал. Направити два модела: један функционални, који не троши време и један који апроксимира број потребних циклуса за извршавање. Направити тестбенч за овај хијерархијски канал. Тестбенч треба да садржи референтни модел са којим се пореди имплементација хијерархијским каналом.

2. Моделовати генератор Фибоначијевих бројава као хијерархијски канал. Направити два модела: један функционални, који не троши време и један који апроксимира број потребних циклуса за извршавање. Направити тестбенч за овај хијерархијски канал. Тестбенч треба да садржи референтни модел са којим се пореди имплементација хијерархијским каналом.