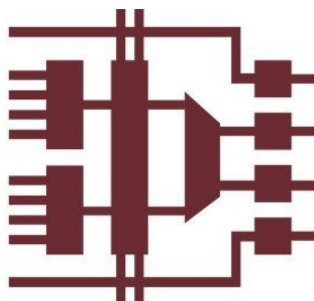
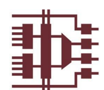


FPGA-SYSTEMS.RU

сообщество FPGA разработчиков



UVM тест таблицы sin/cos

Автор: **Лотник В.Е.**

2020 г.

Содержание

Содержание 1

Список условных обозначений, сокращений и терминов 2

1 Аннотация 3

2 Описание тестируемого компонента 4

3 Описание тестового окружения 6

3.1 Верхний уровень 6

3.2 Тест 7

3.3 Транзакции 9

3.4 Драйвер 11

3.5 Монитор 12

3.6 Scoreboard 13

3.7 Агент 15

3.8 Генератор транзакций 16

3.9 Итог 17

4 Послесловие 19

Список условных обозначений, сокращений и терминов

DUT – Design Under Test

HDL – Hardware Description Language

TLM – Transaction-Level Modeling

UVM – Universal Verification Methodology

VHDL – Very High Speed Integrated Circuit Hardware Description Language

1 Аннотация

В данном руководстве описывается пример построения тестового окружения с использованием UVM для проверки компонента, описанного при помощи HDL.

В качестве тестируемого компонента (DUT) используется таблица синуса/косинуса, описанная на языке VHDL.

Схема подключения тестируемого компонента к тестовому окружению показана на рисунке 1.1.

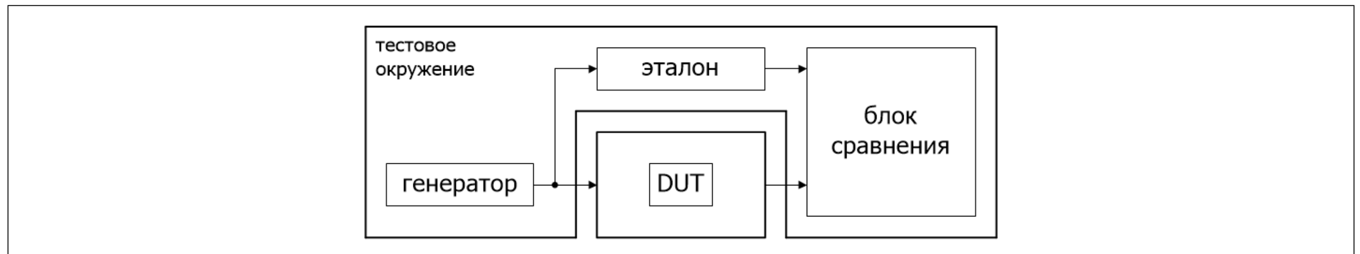


Рисунок 1.1 – Схема тестового окружения

В простейшем случае тестовое окружение содержит в себе:

- генератор входных воздействий для подачи сигнала на DUT;
- некий эталон, функциональность которого должен реализовывать DUT;
- блок сравнения результатов, полученных с эталона и с выхода DUT.

По результатам работы тестового окружения делается вывод о том, насколько тестируемый компонент соответствует эталонной модели.

2 Описание тестируемого компонента

Тестируемый компонент представляет собой табличную реализацию функций $sin(x), cos(x)$ в целочисленной арифметике. Графики данных функций представлены на рисунке 2.1.

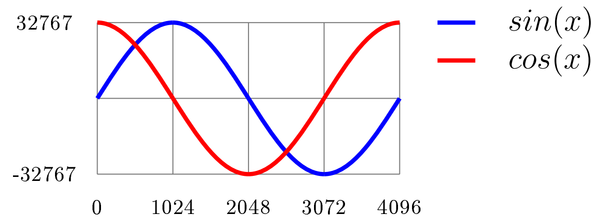


Рисунок 2.1 – Графики функций $sin(x), cos(x)$

Интерфейс DUT представлен в таблице 2.1.

Таблица 2.1 – Порты DUT

Name	Dir	Type	Description
iCLK	**in**	std_logic	тактовый сигнал
iPHASE_V	**in**	std_logic	входная значимость
iPHASE	**in**	[12]unsigned	фаза
oSINCOS_V	**out**	std_logic	выходная значимость
oSIN	**out**	[16]signed	синус
oCOS	**out**	[16]signed	косинус

Вход фазы **iPHASE**, сопровождаемый значимостью **iPHASE_V**, представляет собой 12-битное беззнаковое число в диапазоне $[0 \dots 4095]$.

Выходы синуса **oSIN** и косинуса **oCOS** представляют собой 16-битные знаковые числа в диапазоне $[-32768 \dots 32767]$ и сопровождаются значимостью **oSINCOS_V**.

Временная диаграмма работы DUT представлена на рисунке 2.2.

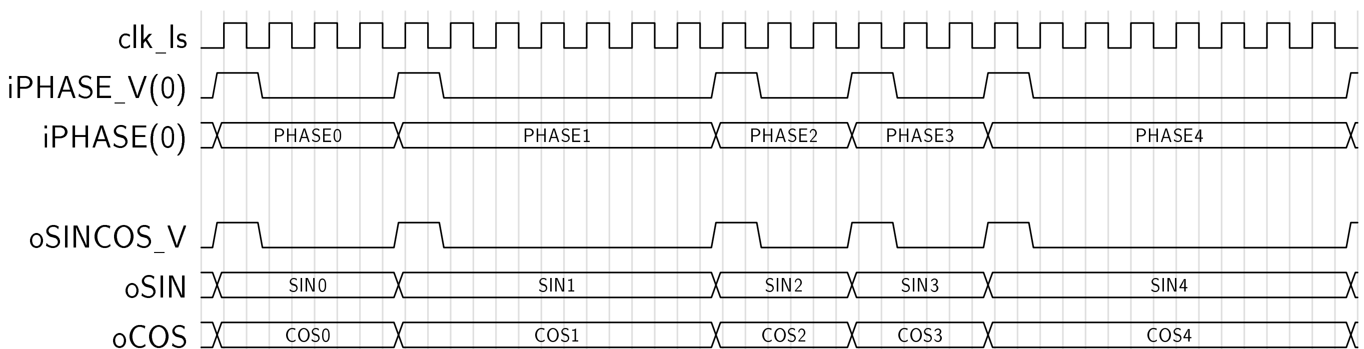


Рисунок 2.2 – Временная диаграмма DUT

> ☞ Приведенная диаграмма не учитывает задержку между приемом и выдачей данных, а также особенности тактирования

Функциональная схема тестируемого компонента показана на рисунке 2.3.

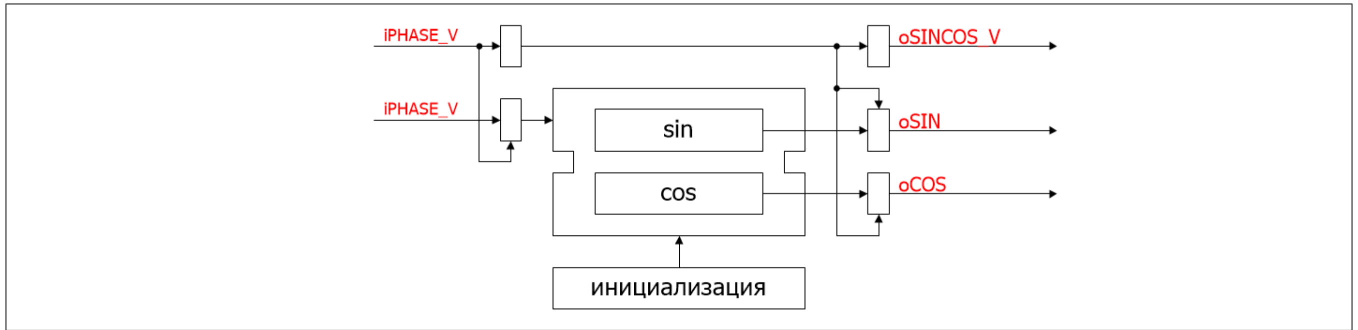


Рисунок 2.3 – Функциональная схема DUT

Ядро компонента – память, в которой хранятся значения синуса/косинуса. Для заполнения памяти используется функция инициализации.

Помимо этого в компоненте используются триггеры для входных и выходных сигналов. Сигнал значимости транслируется со входа на выход с учетом используемых триггеров.

3 Описание тестового окружения

3.1 Верхний уровень

Верхним уровнем тестового окружения в данном примере является файл **tb_top.sv**. Создаем файл с таким названием и следующим содержанием:

```
// tb_top.sv
`timescale 100ps/100ps

module tb_top;
    bit clk = 0;                                          // simple clock
    always #5 clk = ~clk;                                 // 100 MHz
endmodule
```

Здесь нет ничего интересного, кроме объявленного тактового сигнала.

Сюда необходимо подключить тестируемый компонет, для этого создадим файл **sincos_if.sv**, в котором опишем следующий интерфейс:

```
// sincos_if.sv
interface sincos_if (input bit iclk);
    bit                     iphase_v;
    bit[11:0]               iphase;
    bit                     osincos_v;
    bit[15:0]               osin;
    bit[15:0]               ocos;
endinterface
```

Данный интерфейс схож с интерфейсом тестируемого компонента (см. 2). Дополним файл **tb_top.sv** кодом подключения DUT:

```
    sincos_if sincos_if_h(clk);                           // connect iclk to clk

    sin_cos_table #(
    )
    dut(
          .iCLK                 (sincos_if_h.iclk)
        , .iPHASE_V             (sincos_if_h.iphase_v)
        , .iPHASE               (sincos_if_h.iphase)
        , .oSINCOS_V            (sincos_if_h.osincos_v)
        , .oSIN                 (sincos_if_h.osin)
        , .oCOS                 (sincos_if_h.ocos)
    );
```

☞ Особенности сопряжения интерфейсов при симуляции смешанных (VHDL/Verilog) исходников читайте в документации к используемому вами симулятору.

На текущем этапе наше тестовое окружение имеет вид, показанный на рисунке 3.1.

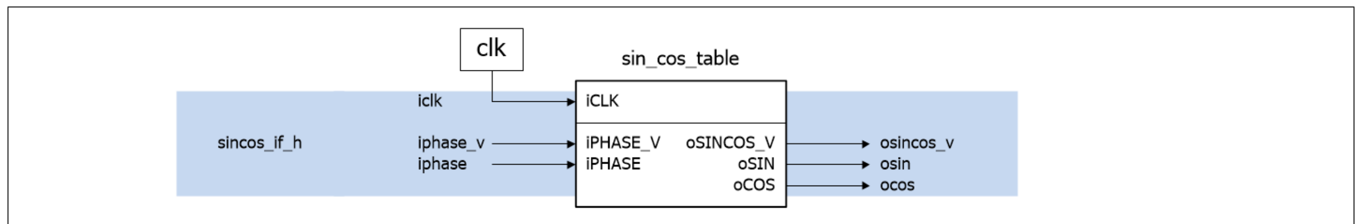


Рисунок 3.1 – Тестовое окружение

На одном клоке далеко не уедешь, поэтому самое время добавить немножко UVM.

3.2 Тест

Первое, что необходимо сделать для создания тестового окружения по UVM, это создать тест. Для этого используется класс **uvm_test**.

Создадим файл **sincos_test_default.svh** со следующим содержанием:

```
// sincos_test_default.svh
class sincos_test_default extends uvm_test;                    // [UVM] class
    `uvm_component_utils(sincos_test_default)                  // [UVM] macro

    extern function new(string name, uvm_component parent);
endclass

function sincos_test_default::new(string name, uvm_component parent);
    super.new(name, parent);
endfunction
```

Здесь для нашего теста объявлен класс, наследующий класс **uvm_test**. Чтобы тест можно было запустить, в нем, как минимум, должен быть объявлен конструктор (функция new()).

☞ Поскольку это ООП, то во всех классах должна быть реализована функция-конструктор класса.

☞ Здесь и далее комментарии с метками [UVM] отмечают код, использующий ресурсы UVM библиотек. В данном руководстве я не буду объяснять, что этот код делает, его просто нужно вставлять. Для полного понимания смысла жизни RTFM по библиотекам UVM.

Для удобства подключения UVM компонентов, которые мы будем реализовывать, создадим пакет **sincos_package.sv** и добавим в него наш тест:

```
// sincos_package.sv
package sincos_package;
    import uvm_pkg::*;                                         // [UVM] package
    `include "uvm_macros.svh"                                  // [UVM] package

    `include "sincos_test_default.svh"
endpackage
```

Теперь наш тест можно запустить из созданного ранее тестового окружения. Для этого дополним файл **tb_top.sv** следующим кодом:

```
1      import uvm_pkg::*;                                     // [UVM] package
2      `include "uvm_macros.svh"                              // [UVM] macroses
3      import sincos_package::*;                              // connect our package
4
5      initial begin
6          run_test("sincos_test_default");                   // [UVM] run test routine
7      end
```

На текущем этапе наше тестовое окружение имеет вид, показанный на рисунке 3.2.

Рисунок 3.2 – Тестовое окружение с пустым тестом

Здесь мы видим наш тестируемый компонент, подключенный через интерфейс; тактовый сигнал, поступающий на этот интерфейс и пустой тест, который никак не взаимодействует с DUT.

Чтобы связать тестируемый компонент с тестом, нужно передать в тест интерфейс, к которому подключен DUT. Для этого модернизируем файлы **tb_top.sv** и **sincos_test_default.svh**:

```
1  // tb_top.sv
2      initial begin
3          uvm_config_db #(virtual sincos_if)::set(           // [UVM] pass interface
4              null, "*", "sincos_if_h", sincos_if_h);        //  to UVM database
5          run_test("sincos_test_default");                   // [UVM] run test routine
6      end
```

```
1  // sincos_test_default.svh
2  class sincos_test_default extends uvm_test;                // [UVM] class
3      `uvm_component_utils(sincos_test_default)              // [UVM] macro
4
5      extern function new(string name, uvm_component parent);
6      extern function void build_phase(uvm_phase phase);     // [UVM] build phase
7
8      virtual sincos_if sincos_if_h;
9  endclass
10
11 function void sincos_test_default::build_phase(uvm_phase phase);
12     // get bfm from database
13     if (!uvm_config_db #(virtual sincos_if)::get(          // [UVM] try to get interface
14         this, "", "sincos_if_h", sincos_if_h)              // from uvm database
15     ) `uvm_fatal("BFM", "Failed to get bfm");              // otherwise throw error
16 endfunction
```

После этого тестовое окружение примет вид, показанный на рисунке 3.3.

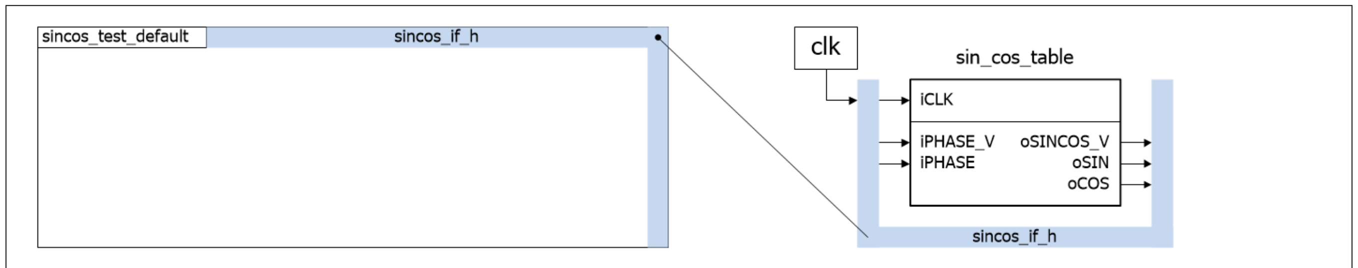


Рисунок 3.3 – Тестовое окружение с пустым тестом и подключенным интерфейсом

Теперь мы можем запускать пустой UVM тест и у него есть связь с нашим тестируемым компонентом.

3.3 Транзакции

Для дальнейшего описания тестового окружения нам нужно определить в каком виде по нашему тесту будут передаваться данные, как из них будут формироваться входные воздействия на DUT, в каком виде будут считываться выходные сигналы, с чем будет работать эталонная модель и так далее.

Для упрощения этих задач в UVM существуют транзакции, для описания которых используется класс **uvm_sequence_item**.

Создадим файл **sincos_seqi.svh**, в котором опишем транзакцию для нашего примера:

```
1  // sincos_seqi.svh
2  class sincos_seqi extends uvm_sequence_item;                    // [UVM] class
3      `uvm_object_utils(sincos_seqi);                              // [UVM] macro
4
5      extern function new(string name = "sincos_seqi");
6
7      rand int phase_v[];
8      rand int phase;
9      int sin;
10     int cos;
11
12     constraint c_phase_v {
13         foreach(phase_v[i])
14             phase_v[i] inside {[0:1]};
15         phase_v.size inside {[5:5]};
16         phase_v.sum == 1;
17     }
18
19     constraint c_phase {
20         phase inside{[0:4095]};
21     }
22
23     extern function bit do_compare(uvm_object rhs, uvm_comparer comparer);
24     extern function string convert2string();
25 endclass
```

Здесь объявлен класс, наследующий класс **uvm_sequence_item**.

В транзакции используются следующие переменные:

- phase_v – массив, предназначенный для формирования сигнала значимости. Данная переменная объявлена как «rand» для возможности использования рандомизации при создании новых транзакций;
- phase – предназначена для передачи значений фазы. Также объявлена как «rand»;
- sin – предназначена для передачи значений синуса;
- cos – предназначена для передачи значений синуса.

Чтобы ограничить значения, которые могут принимать переменные при рандомизации, используются ограничения c_phase_v (ограничивает размер массива и гарантирует, что только один из его элементов будет равен единице) и c_phase (ограничивает значения фаз).

Помимо этого в транзакции реализованы две функции: do_compare() и convert2string().

Функция do_compare() имеет следующую реализацию:

```
function bit sincos_seqi::do_compare(uvm_object rhs, uvm_comparer comparer);
    sincos_seqi RHS;
    bit same;

    same = super.do_compare(rhs, comparer);                // [UVM] call papa

    $cast(RHS, rhs);
    same = (phase == RHS.phase && sin == RHS.sin && cos == RHS.cos) && same;
    return same;
endfunction
```

Функция предназначена для сравнения двух транзакций и возвращает 1, если они равны. В противном случае – 0.

Функция convert2string() имеет следующую реализацию:

```
function string sincos_seqi::convert2string();
    string s;
    s = $sformatf("phase = %6d; sin = %6d, cos = %6d", phase, sin, cos);
    return s;
endfunction
```

Функция предназначена для представления транзакции в виде строки для дальнейшего использования, например, в выводе логов.

Теперь нужно добавить описанную транзакцию в пакет **sincos_package.sv**:

```
    `include "sincos_seqi.svh"
    typedef uvm_sequencer #(sincos_seqi) sincos_seqr;       // [UVM] sequencer
```

Здесь же добавляем **uvm_sequencer**, который будет работать с нашими транзакциями.

3.4 Драйвер

Теперь, когда у нас есть описание транзакций для нашего теста, можно рассмотреть компонент, который будет преобразовывать их в сигналы описанного ранее интерфейса. Для реализации такого компонента используется класс **uvm_driver** – драйвер.

Создадим файл **sincos_drvr.svh**, в котором опишем драйвер для нашего примера:

```
// sincos_drvr.svh
class sincos_drvr extends uvm_driver #(sincos_seqi);          // [UVM] class
    `uvm_component_utils(sincos_drvr)                          // [UVM] macro

    extern function new(string name, uvm_component parent);
    extern task run_phase(uvm_phase phase);                    // [UVM] run phase

    virtual sincos_if sincos_if_h;                             // our interface
    sincos_seqi sincos_seqi_h;                                 // handler for transactions
endclass

task sincos_drvr::run_phase(uvm_phase phase);
    forever begin
        seq_item_port.get_next_item(sincos_seqi_h);            // [UVM] request transaction

            foreach(sincos_seqi_h.phase_v[i]) begin
                @(posedge sincos_if_h.iclk)
                sincos_if_h.iphase_v <= sincos_seqi_h.phase_v[i];
                if (sincos_seqi_h.phase_v[i] == 1'b1)
                    sincos_if_h.iphase <= sincos_seqi_h.phase[11:0];
            end

        seq_item_port.item_done();                             // [UVM] finish transaction
    end
endtask
```

Здесь объявлен класс, наследующий класс **uvm_driver**. Класс содержит интерфейс **sincos_if_h** и драйвер взаимодействует с ним в процессе выполнения run_phase.

Так же в run_phase реализуется механизм TLM: осуществляется запрос транзакции, ее обработка и завершение.

Получив очередную транзакцию, драйвер выполняет цикл для каждого бита из массива phase_v. Переключение элементов массива происходит по тактовому сигналу интерфейса sincos_if_h, при этом каждый бит выставляется на вход iphase_v. При обнаружении ненулевого бита, драйвер выставляет значение фазы phase транзакции на вход iphase интерфейса.

Добавляем драйвер в пакет **sincos_package.sv**:

```
    `include "sincos_drvr.svh"
```

3.5 Монитор

Компонент монитор, по сути, выполняет функцию, противоположную драйверу: он анализирует сигналы интерфейса и на основе анализа формирует транзакции. Для реализации такого компонента используется класс **uvm_monitor**.

Создадим файл **sincos_mont.svh**, в котором опишем монитор для нашего примера:

```
// sincos_mont.svh
class sincos_mont extends uvm_monitor;                      // [UVM] class
    `uvm_component_utils(sincos_mont);                      // [UVM] macro

    extern function new(string name, uvm_component parent);
    extern function void build_phase(uvm_phase phase);      // [UVM] build phase
    extern task run_phase(uvm_phase phase);                 // [UVM] run phase

    virtual sincos_if               sincos_if_h;            // our interface

    sincos_aprt                     sincos_aprt_i;          // analysis port, input
    sincos_seqi                     sincos_seqi_i;          // transaction, input
    sincos_aprt                     sincos_aprt_o;          // analysis port, output
    sincos_seqi                     sincos_seqi_o;          // transaction, output
endclass

function void sincos_mont::build_phase(uvm_phase phase);
    // build analysis ports
    sincos_aprt_i = new("sincos_aprt_i", this);
    sincos_aprt_o = new("sincos_aprt_o", this);
endfunction

task sincos_mont::run_phase(uvm_phase phase);
    forever @(posedge sincos_if_h.iclk) begin
        if (sincos_if_h.iphase_v == 1) begin
            sincos_seqi_i = sincos_seqi::type_id::create("sincos_seqi_i");
            sincos_seqi_i.phase = sincos_if_h.iphase;
            sincos_aprt_i.write(sincos_seqi_i);             // [UVM] write to aprt
        end

        if (sincos_if_h.osincos_v == 1) begin
            sincos_seqi_o = sincos_seqi::type_id::create("sincos_seqi_o");
            sincos_seqi_o.sin = $signed(sincos_if_h.osin);
            sincos_seqi_o.cos = $signed(sincos_if_h.ocos);
            sincos_aprt_o.write(sincos_seqi_o);             // [UVM] write to aprt
        end
    end
endtask
```

Здесь объявлен класс, наследующий класс **uvm_monitor**. Класс содержит интерфейс **sincos_if_h** и монитор взаимодействует с ним в процессе выполнения run_phase.

По тактовому сигналу монитор проверяет входную (iphase_v) значимость интерфейса sincos_if_h. Если она не равна нулю, то формируется новая входная транзакция (sincos_seqi_i), которая записывается в порт анализа (sincos_aprt_i).

Аналогично, по выходной значимости (osincos_v), формируется выходная транзакция (sincos_seqi_o), записываемая в (sincos_aprt_o).

Добавляем монитор в пакет **sincos_package.sv**:

```
    typedef uvm_analysis_port #(sincos_seqi) sincos_aprt;
    `include "sincos_mont.svh"
```

Здесь же добавляем **uvm_analysis_port**, который будет работать с нашими транзакциями.

3.6 Scoreboard

Поскольку прямой перевод слова «scoreboard» (табло) здесь не очень подходит, далее будет использоваться английское написание.

Данный компонент выполняет две функции:

- формирование эталонной транзакции на основе данных, полученных из анализа входных сигналов интерфейса **sincos_if_h**;
- сравнивание эталонной транзакции с данными, полученными из анализа выходных сигналов интерфейса **sincos_if_h**.

Создадим файл **sincos_scrb.svh**, в котором опишем scoreboard для нашего примера:

```
// sincos_scrb.svh
`uvm_analysis_imp_decl(_i)                                      // [UVM] macro
`uvm_analysis_imp_decl(_o)                                      // [UVM] macro

class sincos_scrb extends uvm_scoreboard;                       // [UVM] class
    `uvm_component_utils(sincos_scrb)                           // [UVM] macro

    extern function new(string name, uvm_component parent);
    extern function void build_phase(uvm_phase phase);          // [UVM] build phase

    uvm_analysis_imp_i #(sincos_seqi, sincos_scrb) sincos_aprt_i;
    uvm_analysis_imp_o #(sincos_seqi, sincos_scrb) sincos_aprt_o;

    sincos_seqi sincos_seqi_queue_i[$];
    sincos_seqi sincos_seqi_queue_o[$];

    extern virtual function void write_i(sincos_seqi sincos_seqi_h);
    extern virtual function void write_o(sincos_seqi sincos_seqi_h);

    extern function void processing();

    extern virtual function int get_ideal_sin(int phase, int max = (2 ** 15 - 1));
    extern virtual function int get_ideal_cos(int phase, int max = (2 ** 15 - 1));
endclass
```

Здесь объявлен класс, наследующий класс **uvm_scoreboard**. Класс содержит порты анализа для входных (sincos_aprt_i) и выходных (sinco_aprt_o) транзакций. При записи данных в эти порты вызываются функции write_i() и write_o() соответственно:

```
function void sincos_scrb::write_i(sincos_seqi sincos_seqi_h);
    sincos_seqi_queue_i.push_back(sincos_seqi_h);
endfunction

function void sincos_scrb::write_o(sincos_seqi sincos_seqi_h);
    sincos_seqi_queue_o.push_back(sincos_seqi_h);
    processing();
endfunction
```

При вызове функции write_i входная транзакция записывается в очередь sincos_seqi_queue_i.

При вызове функции write_o выходная транзакция записывается в очередь sincos_seqi_queue_o, после чего вызывается функция processing():

```
function void sincos_scrb::processing();
    sincos_seqi sincos_seqi_i;
    sincos_seqi sincos_seqi_o;
    string data_str;

    sincos_seqi_i = sincos_seqi_queue_i.pop_front();
    sincos_seqi_i.sin = get_ideal_sin(sincos_seqi_i.phase);
    sincos_seqi_i.cos = get_ideal_cos(sincos_seqi_i.phase);

    sincos_seqi_o = sincos_seqi_queue_o.pop_front();
    sincos_seqi_o.phase = sincos_seqi_i.phase;

    data_str = {
            "\n", "actual:    ", sincos_seqi_o.convert2string(),
            "\n", "predicted: ", sincos_seqi_i.convert2string()
        };

    if (!sincos_seqi_i.compare(sincos_seqi_o)) begin
        `uvm_error("FAIL", data_str)
        fail_cnt++;
    end else
        `uvm_info("PASS", data_str, UVM_HIGH)
endfunction
```

Из очереди входных транзакций считывается sincos_seqi_i и для ее значения фазы phase рассчитываются эталонные значения sin/cos путем вызова функций get_ideal_sin()/get_ideal_cos().

Из очереди выходных транзакций считывается sincos_seqi_o и ее значение фазы phase копируется из входной транзакции (для простоты сравнения).

После этого формируется строка для вывода значений двух транзакций, выполняется их сравнение и результат выводится в лог.

Таким образом, в простейшем случае, можно оценить результаты работы теста путем чтения лога.

Добавляем scoreboard в пакет **sincos_package.sv**:

```
    `include "sincos_scoreboard.svh"
```

3.7 Агент

Компоненты класса **uvm_agent** используются для группировки других компонентов, работающих с одним интерфейсом. Для нашего примера мы сгруппируем написанные ранее драйвер, монитор и scoreboard.

Создадим файл **sincos_agnt.svh**:

```
// sincos_agnt.svh
class sincos_agnt extends uvm_agent;                        // [UVM] class
    `uvm_component_utils(sincos_agnt)                       // [UVM] macro

    extern function new(string name, uvm_component parent);
    extern function void build_phase(uvm_phase phase);      // [UVM] build phase
    extern function void connect_phase(uvm_phase phase);    // [UVM] connect phase

    virtual sincos_if sincos_if_h;                          // our interface

    sincos_seqr                     sincos_seqr_h;
    sincos_drvr                     sincos_drvr_h;
    sincos_mont                     sincos_mont_h;
    sincos_scrb                     sincos_scrb_h;
endclass

function void sincos_agnt::build_phase(uvm_phase phase);
    sincos_seqr_h = uvm_sequencer #(sincos_seqi)::type_id::create("sincos_seqr_h", this);
    sincos_drvr_h = sincos_drvr::type_id::create("sincos_drvr_h", this);
    sincos_mont_h = sincos_mont::type_id::create("sincos_mont_h", this);
    sincos_scrb_h = sincos_scrb::type_id::create("sincos_scrb_h", this);

    sincos_drvr_h.sincos_if_h = this.sincos_if_h;
    sincos_mont_h.sincos_if_h = this.sincos_if_h;
endfunction

function void sincos_agnt::connect_phase(uvm_phase phase);
    sincos_drvr_h.seq_item_port.connect(sincos_seqr_h.seq_item_export);
    sincos_mont_h.sincos_aprt_i.connect(sincos_scrb_h.sincos_aprt_i);
    sincos_mont_h.sincos_aprt_o.connect(sincos_scrb_h.sincos_aprt_o);
endfunction
```

Здесь объявлен класс, наследующий класс **uvm_agent**. Класс содержит интерфейс **sincos_if_h**, секвенсер, драйвер, монитор и scoreboard.

В функции connect_phase интерфейс передается в драйвер и монитор, драйвер подключается к секвенсеру, порты анализа монитора подключаются к соответствующим портам анализа scoreboard.

Полученный агент имеет структуру, представленную на рисунке 3.4.

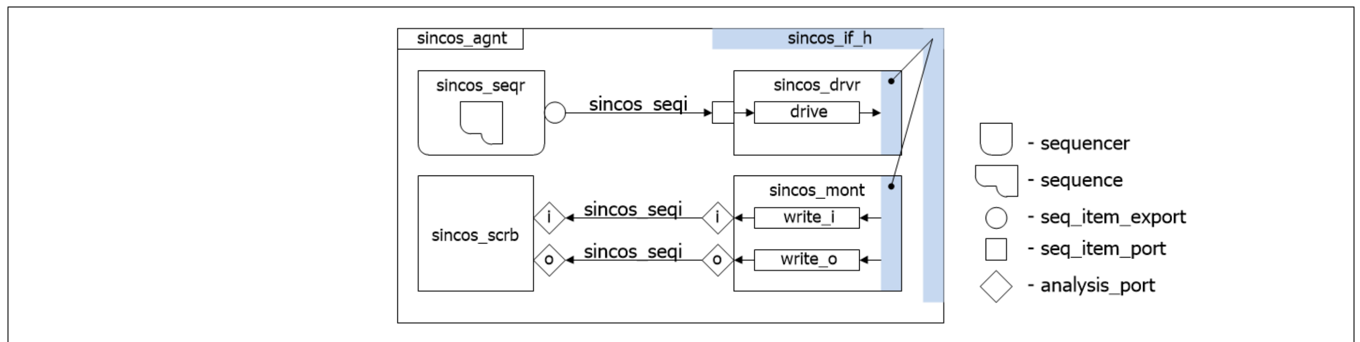


Рисунок 3.4 – Структура класса-агента

Не забываем добавить агент в пакет **sincos_package.sv**:

```
'include "sincos_agnt.svh"
```

Осталось встроить полученный компонент в тест, который мы создали в 3.2 и добавить к нему генератор транзакций (sequence).

3.8 Генератор транзакций

Генератор транзакций предназначен для формирования, очереди транзакций по некоторому алгоритму. Для реализации такого компонента используется класс **uvm_sequence**.

Создадим файл **sincos_seqc_default.svh**, в котором опишем генератор для нашего теста **sincos_test_default**:

```
// sincos_seqc_default.svh
class sincos_seqc_default extends uvm_sequence #(sincos_seqi); // [UVM] class
    'uvm_object_utils(sincos_seqc_default);                     // [UVM] macro

    extern function new(string name = "sincos_seqc_default");
    extern task body();

    sincos_seqi sincos_seqi_h;
endclass

task sincos_seqc_default::body();
    repeat(100) begin
        sincos_seqi_h = sincos_seqi::type_id::create("sincos_seqi_h");
        start_item(sincos_seqi_h);                              // [UVM] start transaction
            assert(sincos_seqi_h.randomize());
        finish_item(sincos_seqi_h);                             // [UVM] finish transaction
    end
endtask
```

Здесь объявлен класс, наследующий класс **uvm_sequence**. Описанный генератор в цикле выполняет следующие действия:

- создает новую транзакцию;
- рандомизирует ее;
- заканчивает транзакцию.

3.9 Итог

Встраиваем описанные нами компоненты в тест, для этого возвращаемся к файлу **sincos_test_default.svh**:

```
// sincos_test_default.svh
class sincos_test_default extends uvm_test;                    // [UVM] class
    `uvm_component_utils(sincos_test_default)                  // [UVM] macro

    extern function new(string name, uvm_component parent);
    extern function void build_phase(uvm_phase phase);         // [UVM] build phase
    extern task run_phase(uvm_phase phase);                    // [UVM] run phase

    virtual sincos_if sincos_if_h;                             // virtual handler

    sincos_agnt                     sincos_agnt_h;
    sincos_seqc_default             sincos_seqc_default_h;
endclass

function void sincos_test_default::build_phase(uvm_phase phase);
    // get bfm from database
    if (!uvm_config_db #(virtual sincos_if)::get(              // [UVM] try get interface
        this, "", "sincos_if_h", sincos_if_h)                  // from uvm database
    ) `uvm_fatal("BFM", "Failed to get bfm");                  // otherwise throw error

    sincos_agnt_h = sincos_agnt::type_id::create("sincos_agnt_h", this);
    sincos_agnt_h.sincos_if_h = this.sincos_if_h;

    sincos_seqc_default_h =
        sincos_seqc_default::type_id::create("sincos_seqc_default_h", this);
endfunction

task sincos_test_default::run_phase(uvm_phase phase);
    phase.raise_objection(this);                               // [UVM] start sequence
        sincos_seqc_default_h.start(sincos_agnt_h.sincos_seqr_h);
    phase.drop_objection(this);                                // [UVM] finish sequence
endtask
```

Здесь мы добавили агент и генератор транзакций. Интерфейс, полученный ранее из верхнего уровня, передается в агент, в результате чего все его компоненты, использующие этот интерфейс, оказываются подключенными к DUT.

Остается только запустить генератор, использовав для него секвенсер, расположенный внутри агента.

В результате наше тестовое окружение имеет вид, показанный на рисунке 3.5.

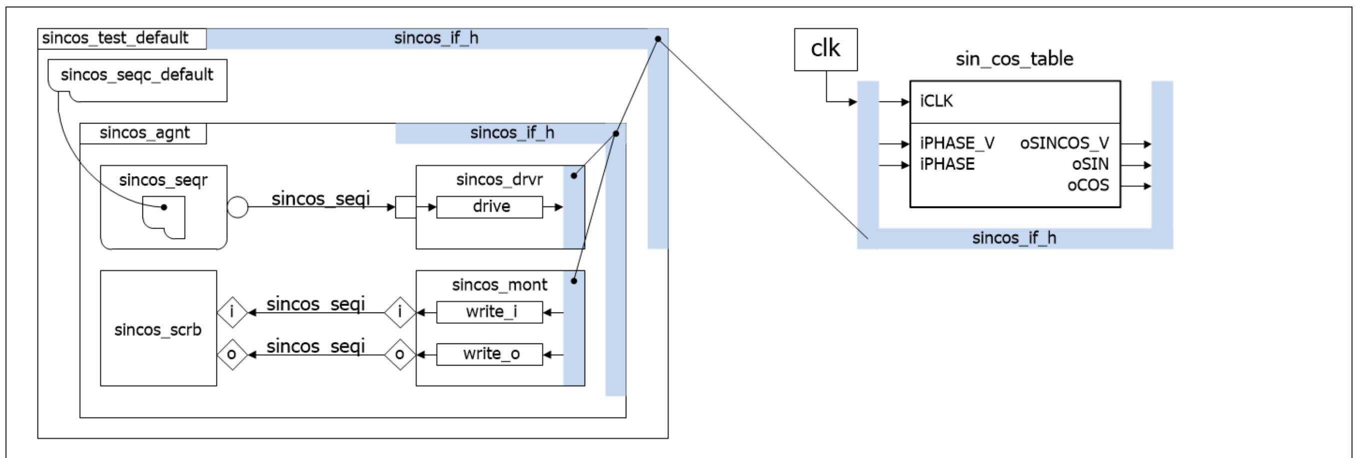


Рисунок 3.5 – Итоговое тестовое окружение

Теперь после запуска теста генератор начнет выдавать транзакции, передавать их в агент, после чего они будут переданы драйверу (по запросу от него). Драйвер сформирует временную диаграмму на вход DUT. DUT выдаст сигналы на выход. Мониторы считают вход и выход тестируемого компонента, сформируют транзакции и передадут их в scoreboard. Scoreboard, в свою очередь, определит, насколько работа DUT соответствует эталону.

4 Послесловие

Исходные коды для рассмотренного примера выложены в **репозиторий**.

Для дальнейшего изучения UVM и подходов к написанию крутых тестовых окружений рекомендую почитать следующие источники:

- Accellera UVM Class Reference Manual 1.2 – официальное описание библиотеки UVM
- Accellera UVM Users Guide 1.2 – обширное описание техник применения компонентов UVM
- Salemi R., The UVM Primer - An Introduction to the Universal Verification Methodology(2013) Собственно, данный пример был создан на основе примеров из этой книги.
- Spear C., SystemVerilog for Verification A Guide to Learning the Testbench Language Features(2012)
- **Verification Academy** – сайт содержит больше количество информации по верификации, для полного доступа нужна регистрация на корпоративную почту
- **Гитхаб** – можно найти примеры

Если вы дочитали до этого момента, то респект.