Верификация проектов

Типы верификации

- Функциональная
  - Моделирование
- Формальная
  - Проверка выполнение системой требований
  - Доказательство эквивалентности реализаций
- Статический анализ кода
  - Детектирование ошибок кодирования
- Физическая
  - Проверка соблюдения технологических норм (для СБИС)
- Временная
  - Проверка соответствия проекта временным требованиям

Функциональная верификация проекта

- Цель
  - Доказать корректность функционирования устройства путем моделирования
- Метод
  - Определить метрики, контролирующие процесс верификации
  - Создание тестового окружения
  - Генерация воздействий
  - Моделирование и контроль результатов
- Средства
  - Симуляторы HDL: Modelsim, Questa, ActiveHDL, RivieraPro, VCS, Incisive,…
  - IP для верификации: **BFM and Monitor Components**

Компоненты и средства

- Avalon Verification IP
  - Методика использования
  - Компоненты библиотеки
  - Организация тестбенча
- Возможности SystemVerilog для верификации
- Методология верификации UVM
  - Назначение
  - Организация
  - Пример

Метрики верификации

- Покрытие кода (code coverage)
  - Строк кода (code coverage)
  - Переключения регистров (toggle coverage)
  - …
- Покрытие утверждений (assertions)
  - System Verilog Assertions (SVA), Property Specification Language (PSL),…
- Функциональное покрытие (functional coverage)
  - Выполнение функций, определенных планом теста

Методы

- Напишем тестбенч, что-то протестируем
  - Что тестируем - правильную работу или реакцию на неправильные воздействия?
  - Достаточно ли покрыли входные воздействия и состояния схемы?
  - Соответствует требованиям к устройству?
  - Соответствует плану тестирования?

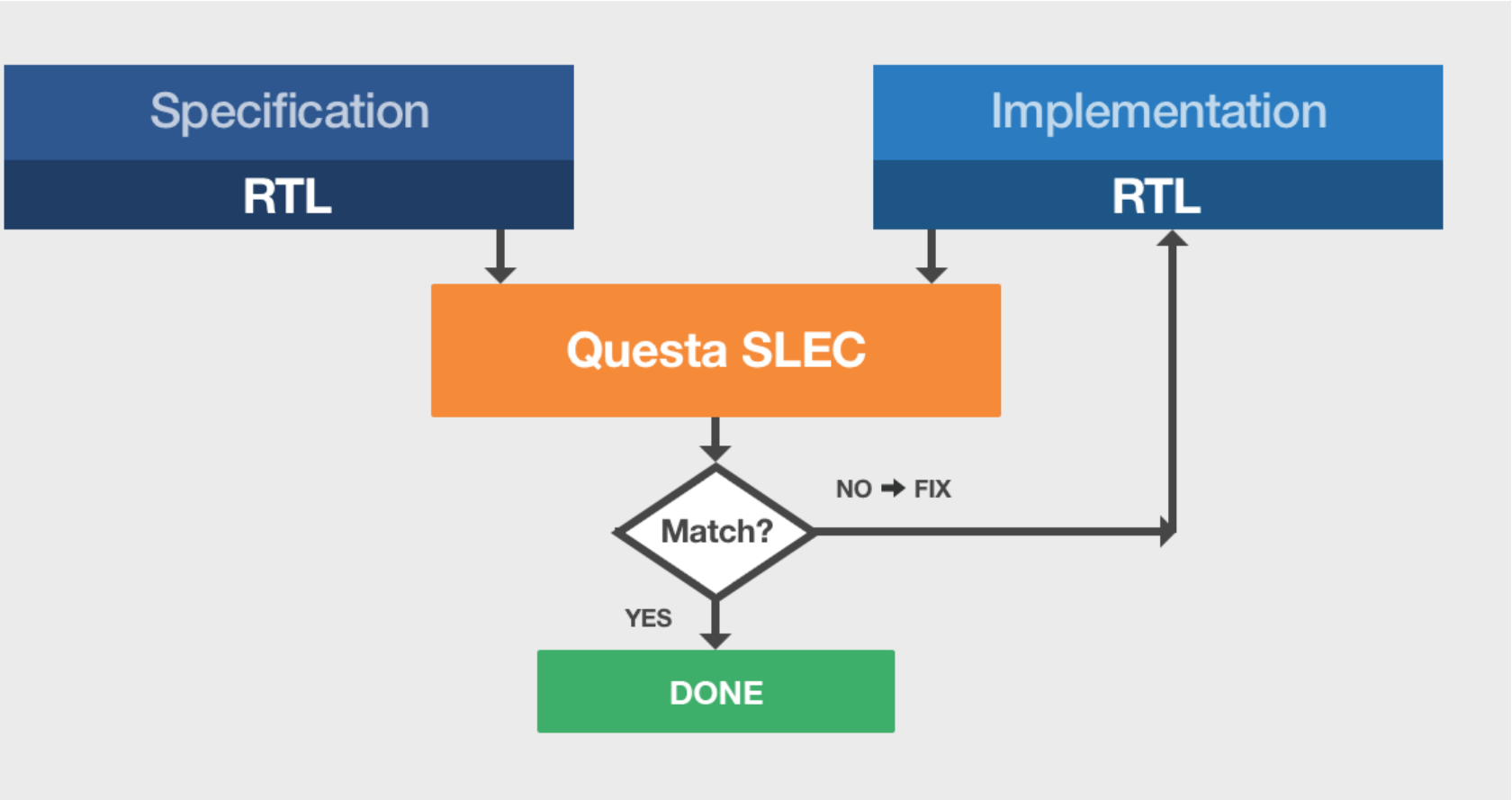
Методы

- Coverage Driven Verification (CDV)
  - Верификация, управляемая покрытием. Требования покрытия (обычно кода) определяют тесты, включаемые в план тестирования.
- Assertion-Based Verification (ABV)
  - Верификация на основе утверждений. Проект дополняется утверждениями свойств (например, SVA), которые проверяются во время моделирования. Требуется обеспечить покрытие утверждений.
- Constrained Random Verification (CRV)
  - Верификация случайными значениями с ограничениями. Использует возможности SystemVerilog по формированию случайных воздействий в определенных диапазонах значений с контролем покрытия воздействий.
- Все это хорошо сочетается вместе

Формальная верификация проекта

- Цель
  - Доказать корректность функционирования устройства путем доказательства свойств
- Методы
  - Доказательство эквивалентности реализаций (equivalence checking)
  - Проверка выполнения системой утверждений (formal property verification)
  - …
- Средства
  - Mentor: Formal Pro, Questa Formal
  - Cadence: Jasper Gold, Conformal LEC
  - Synopsys: VC Formal

Доказательство эквивалентности



Верификация с применением BFM на примере Avalon Verification IP

Компоненты Avalon Verification Suite

- Компоненты BFM
  - Avalon MM Master
  - Avalon MM Slave
  - Avalon ST Source
  - Avalon ST Sink
  - AXI3 Master
  - AXI3 Slave
  - AXI4 Master
  - AXI4 Slave
- Мониторы
  - Avalon MM
  - Avalon ST
  - AXI3 Inline Monitor
  - AXI4 Inline Monitor

- *Все поддерживаемые симуляторы:*
  - *Modelsim AE / SE (Mentor Graphics)*
  - *Questa Simulator (Mentor Graphics)*
  - *Active HDL (Aldec)*
  - *Riviera Pro (Aldec)*
- *Только полнофункциональные версии:*
  - *Questa Simulator (Mentor Graphics)*
  - *Riviera Pro (Aldec)*

Компоненты BFM

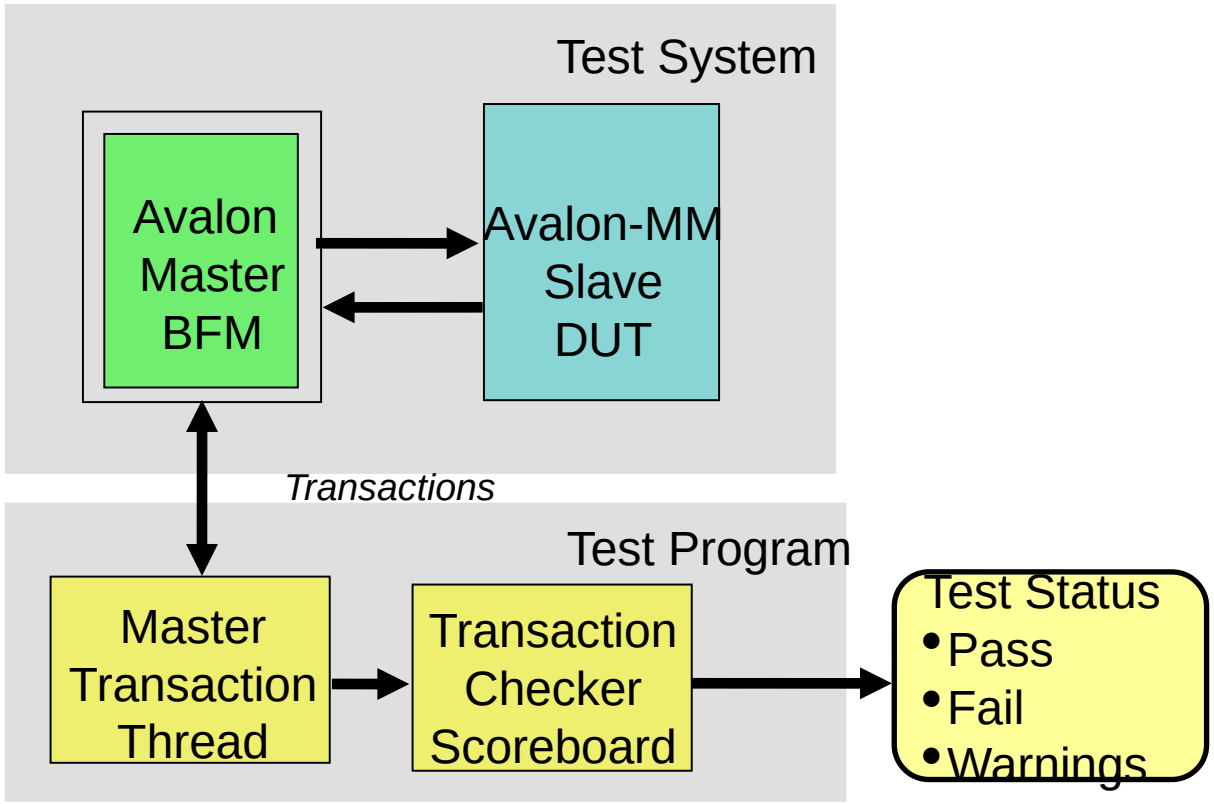
- Компонент Qsys
- Создается экземпляр в тестбенче
- Исполняемая спецификация Avalon
- Транзакции управляют формированием сигналов интерфейса Avalon
- Управляется программой теста в тестбенче
- Также поддерживаются AXI3, AXI4, AXI4 Stream

Применение BFM

- Эмулировать поведение интерфейса на тестируемых компонентах с произвольной конфигурацией интерфейса

- Avalon-MM Masters
  - Processor Memory Interfaces
  - Switch fabric network adapter output
- Avalon-MM Slaves
  - Memory Interface
  - Switch fabric network adapter input
- Avalon-ST Sources
  - Switch fabric node output
  - DSP component output
- Avalon-ST Sinks
  - Switch fabric node input
  - DSP component input

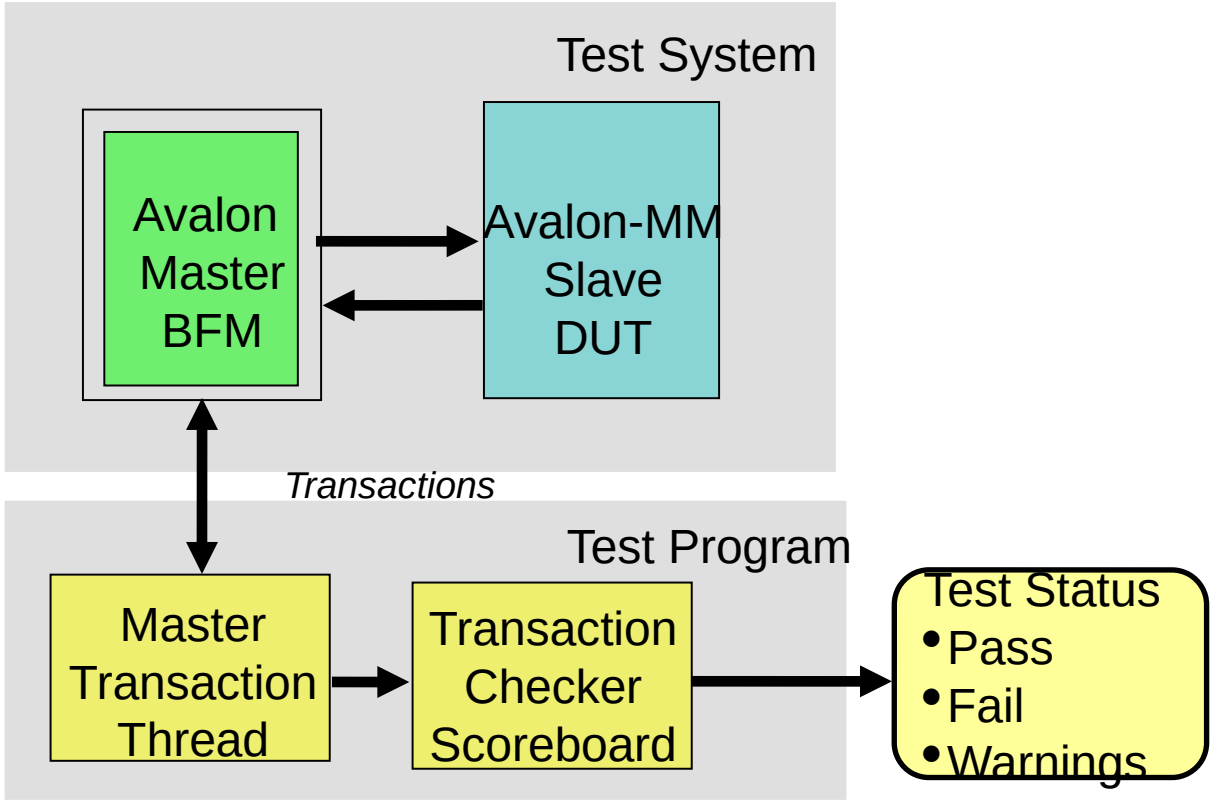
Простой тестбенч



Компоненты монитора

- Компонент Qsys
- Создается экземпляр в тестбенче
- Исполняемая спецификация Avalon
- Контролирует сигналы интерфейса Avalon
  - Исполняет операторы assertions для протокола
  - Измеряет тестовое покрытие
- Конфигурируется программой теста в тестбенче

Простой тестбенч

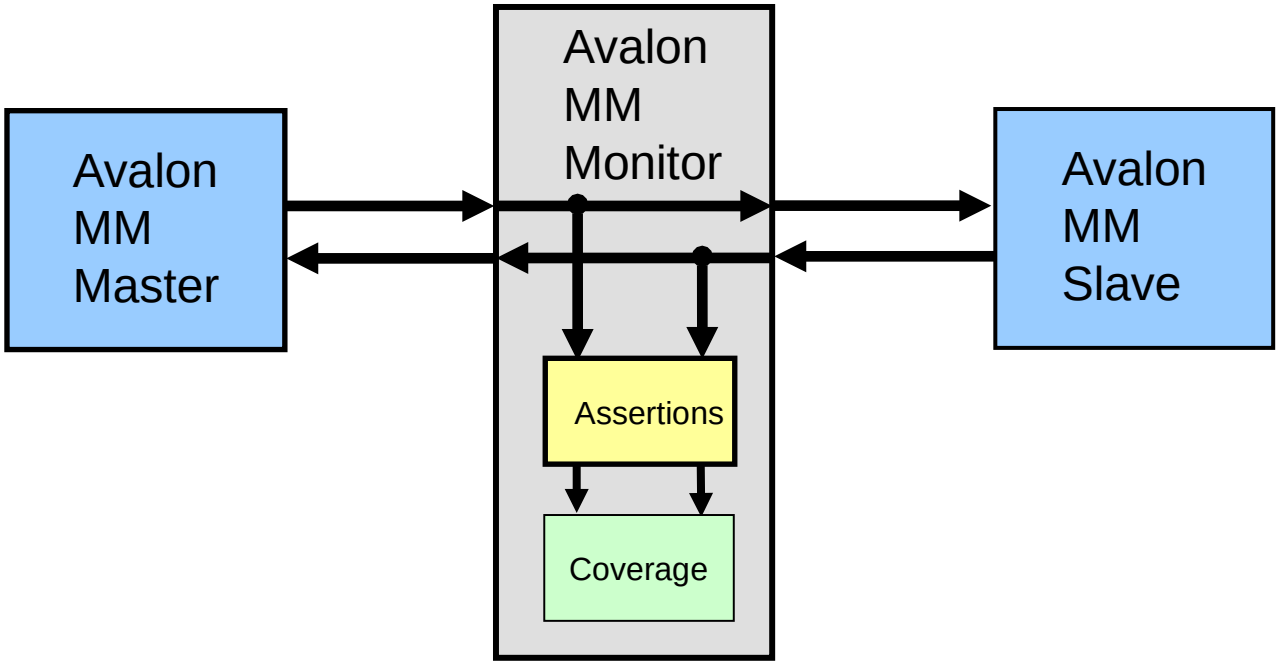


Компоненты монитора

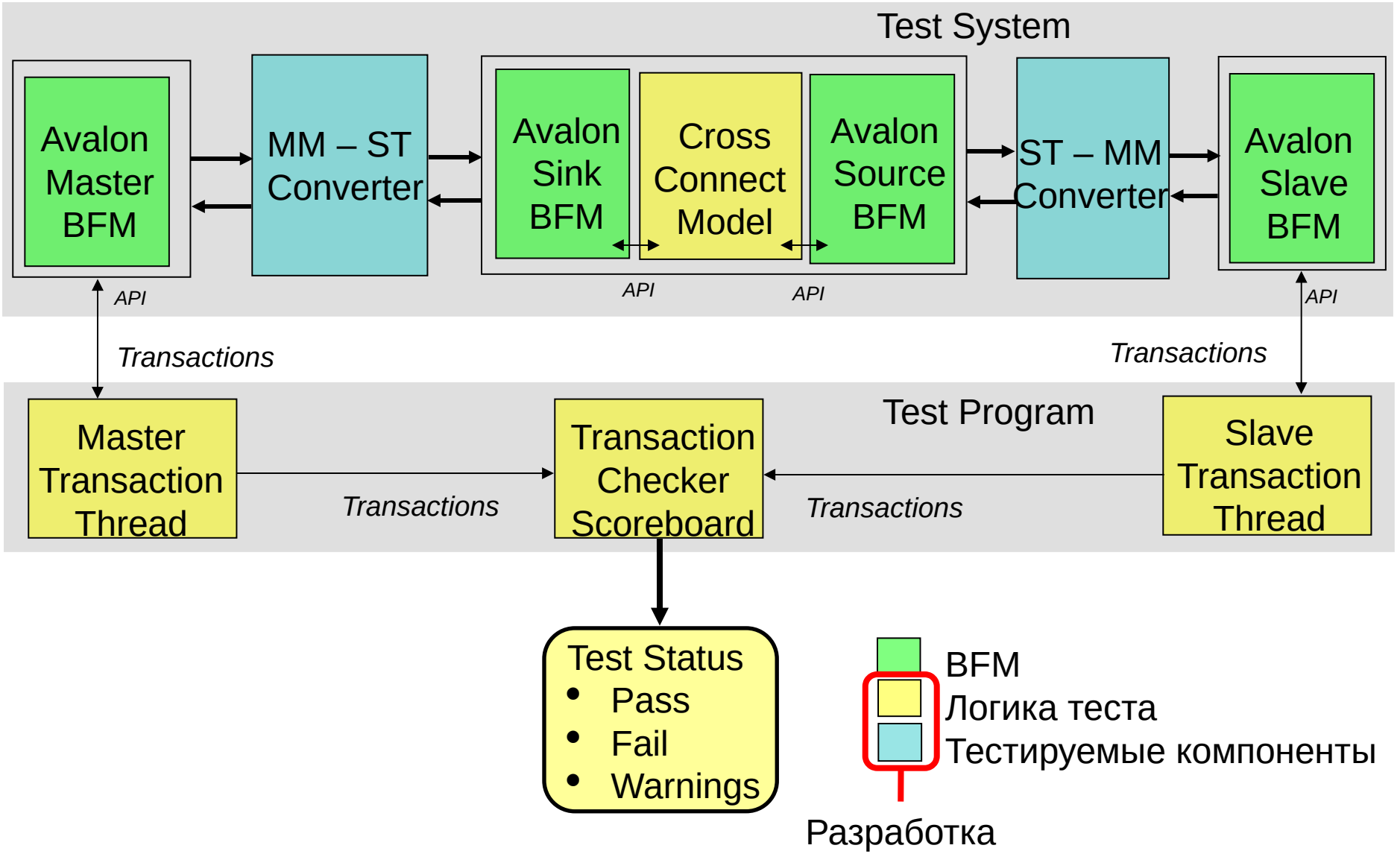
- Компонент Qsys
- Создается экземпляр в тестбенче
- Исполняемая спецификация Avalon
- Контролирует сигналы интерфейса Avalon
  - Исполняет операторы assertions для протокола
  - Измеряет тестовое покрытие
- Конфигурируется программой теста в тестбенче

Монитор

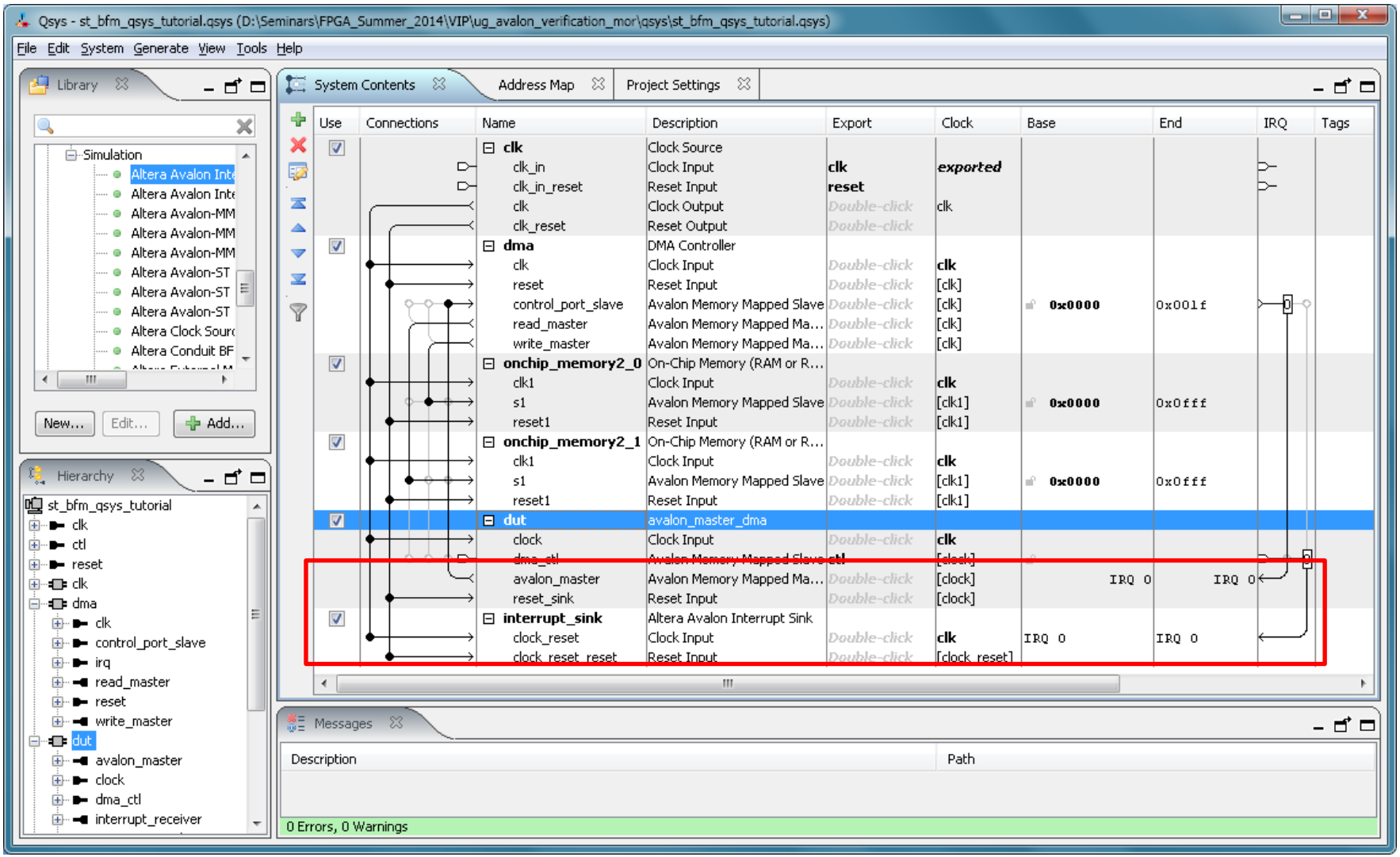
- Это мост
- Передачи пропускаются без изменений
- Данные отводятся для проверки условий и оценки покрытия



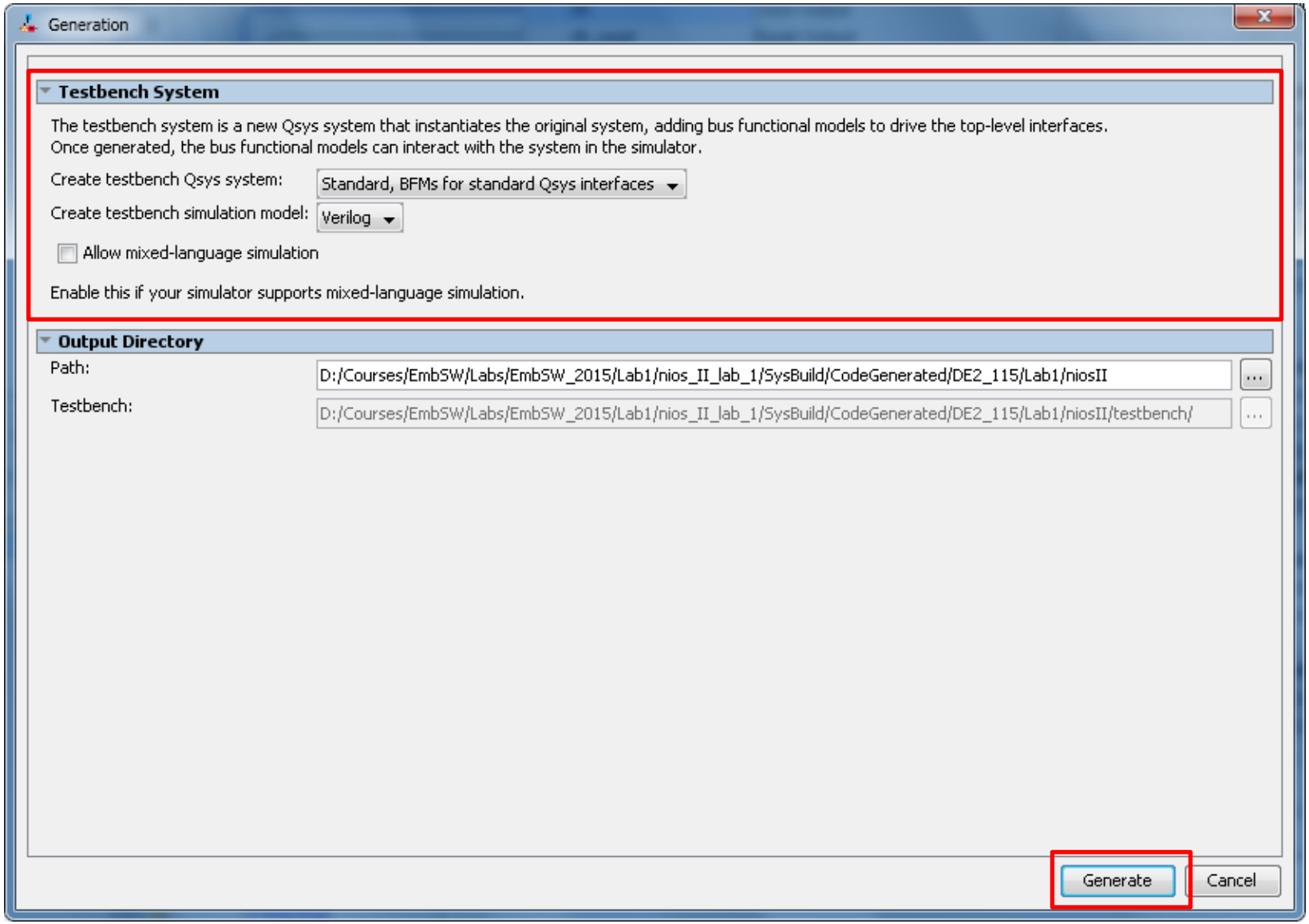
Более сложный тестбенч



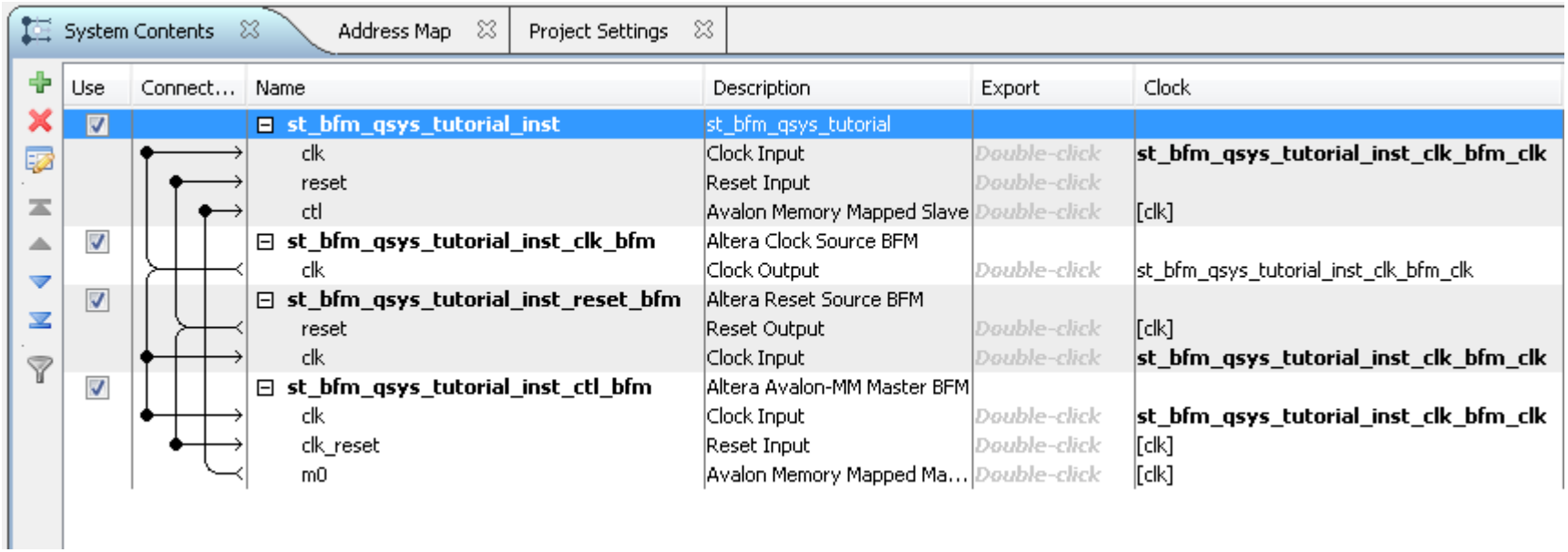
Тестируемый модуль в Qsys (1)



Генерация в Qsys (2)



Система тестбенча в Qsys (3)



Тестовая программа (4)

```
// console messaging level
`define VERBOSITY VERBOSITY_INFO

//BFM hierachy
`define CLK tb.st_bfm_qsys_tutorial_inst_clk_bfm
`define RST tb.st_bfm_qsys_tutorial_inst_reset_bfm
`define SLAVE tb.st_bfm_qsys_tutorial_inst_ctl_bfm
//`define SLAVE tb.st_bfm_qsys_tutorial_inst_master_ctl_bfm_m0_translator

//BFM related parameters
`define AV_ADDRESS_W 3
`define AV_DATA_W 32

module test_program();

  import verbosity_pkg::*;
  import avalon_mm_pkg::*;
  string message;

    // Avalon-MM single-transaction write
    task avalon_write (
        input [`AV_ADDRESS_W-1:0] addr,
        input [`AV_DATA_W-1:0]    data
    );
    begin
        // Construct the SLAVE request
        `SLAVE.set_command_request(REQ_WRITE);
        `SLAVE.set_command_idle(0, 0);
        `SLAVE.set_command_init_latency(0);
        `SLAVE.set_command_address(addr);
        `SLAVE.set_command_byte_enable('1,0);
        `SLAVE.set_command_data(data, 0);

        // Queue the command
        `SLAVE.push_command();

        // Wait until the transaction has completed
        while (`SLAVE.get_response_queue_size() != 1)
            @(posedge `CLK.clk);

        // Dequeue the response and discard
        `SLAVE.pop_response();
    end
    endtask
```

```
  // Avalon-MM single-transaction read
  task avalon_read (
      input  [`AV_ADDRESS_W-1:0] addr,
      output [`AV_DATA_W-1:0]    data
  );
  begin
      // Construct the SLAVE request
      `SLAVE.set_command_request(REQ_READ);
      `SLAVE.set_command_idle(0, 0);
      `SLAVE.set_command_init_latency(0);
      `SLAVE.set_command_address(addr);
      `SLAVE.set_command_byte_enable('1,0);
      `SLAVE.set_command_data(0, 0);

      // Queue the command
      `SLAVE.push_command();

      // Wait until the transaction has completed
      while (`SLAVE.get_response_queue_size() != 1)
          @(posedge `CLK.clk);

      // Dequeue the response and return the data
      `SLAVE.pop_response();
      data = `SLAVE.get_response_data(0);
  end
  endtask

 initial
 begin
          set_verbosity(`VERBOSITY);
  `SLAVE.init();

  //wait for reset to de-assert and trigger start_test event
          wait(`RST.reset == 1);
  avalon_write(1'h0, 32'h00);
  avalon_write(1'h0, 32'h00);
  avalon_write(1'h0, 32'h00);
  avalon_write(3'h1, 32'h40);
  avalon_write(3'h2, 32'h20);
  avalon_write(3'h3, 32'h30);

  avalon_write(3'h6, 32'h2fc);//13'b0000111111100);

 end

endmodule
```

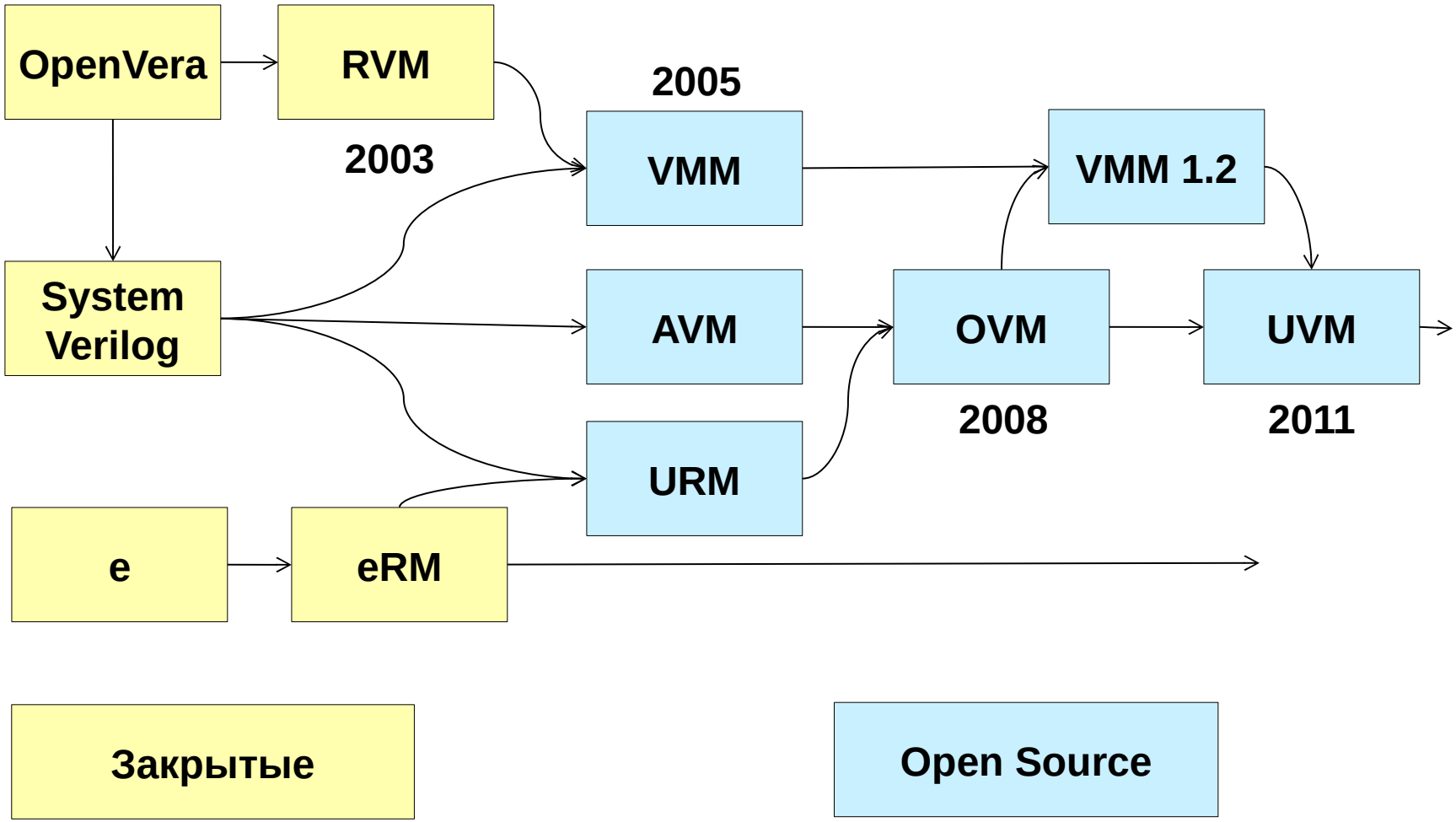
Список IP для верификации

Добавляются в систему из Qsys или автоматически при создании тестбенча для экспортируемых интерфейсов

Работа с BFM описана в Avalon Verification IP Suite User Guide

Методологии верификации UVM/OVM/…

История методологий верификации



SystemVerilog как основа UVM

SystemVerilog

- HDVL - Hardware description and verification language
- Возник как расширение Verilog и Vera
- Ориентирован на описание моделей сложных систем и их верификацию
- Стандарт IEEE1800
  - Текущая версия – IEEE1800-2017, объединяет стандарты Verilog и SystemVerilog
- Содержит большое количество дополнительных возможностей для верификации
  - ООП, интерфейсы, механизм SVA, checkers, constrained random value generation, coverage и т.д.

Объявление интерфейса

Интерфейс – конструкция языка SystemVerilog, которая инкапсулирует обмен данными между модулями. В простейшем случае объединяет проводники и шины в один интерфейс.

```
//объявление интерфейса
interface bus_if;
    logic clock, reset, cs, wr_n;
    logic [1:0] addr;
    logic [31:0] dwr;
    logic [31:0] drd;
endinterface interface dut_if;

//использование интерфейса на верхнем уровне тестбенча
module top;
    bus_if bus_inst();
    master gen(.bus(bus_inst));
    slave dut(.bus(bus_inst));
endmodule

//объявление одного из модулей и использование сигналов
module slave(bus_if bus)
…
    always_ff @(posedge bus.clock, posedge bus.reset)
    begin
         if (bus.reset) …
         else …
    end

endmodule
```

Пакеты

Пакеты позволяют инкапсулировать разные типы данных, константы, функции и т.д.., облегчая их повторное использование в коде, так как определение дается в одном месте.

```
package project_types;
    typedef logic [31:0] data_t;
    typedef enum [3:0] {ADD = 3'b001, SUB = 3'b011, ...} opcodes_t;
    typedef struct {
         opcodes_t opcode;
         data_t data;
    } operation_t;
    function automatic logic parity_gen(data_t d);
         return ^d;
    endfunction
endpackage

module ALU
    import project_types::*;
    (input operation_t operation, output data_t result, output logic parity);
…
    case (operation.opcode)
    ADD:
…
endmodule
```

ООП в SystemVerilog

```
program class_prog;

class packet;
    // члены класса
    int size;
    int data [];
    // конструктор
    function new (int size);
        this.size = size;
        data = new[size];
        for (i=0; i < this.size; i ++)
            data[i] = $random;
    end
    endfunction
    // задача
    task print ();
        for (i=0; i < size; i ++)
        begin
            $write("%x ",data[i]);
            $write("\n");
        end
    endtask
    // функция
    function integer get_size();
        return this.size;
     endfunction
endclass

packet pkt;

initial begin
    pkt = new(5);
    pkt.print();
    $display ("Size of packet
    %0d",pkt.get_size());
end

endprogram
```

Классы поддерживают шаблоны, множественное наследование интерфейсов, могут включать в себя различные объекты – переменные, функции, задачи и т.д.

SVA (SystemVerilog Assertions)

- Определяют поведение системы
- Используются для контроля корректности поведения системы
- Могут группироваться в checkers для одновременной оценки нескольких условий
- Пример:

```
assert (property (@(posedge clk) req |-> ##[3:10] ack;)
else $error;
```

  - Если req – истина по текущему тактовому импульсу, то в интервале от 3 до 10 тактов должен установиться ack
  - Для описания такой зависимости в тестбенче на Verilog потребуется описать счетчик, запускать его по req и контролировать ack в нужном интервале

Constrained random value generation

- Генерация псевдослучайных значений с ограничениями
- Позволяет автоматизировать тестирование
- Пример:

```
class Bus;
   rand bit[15:0] addr;
   rand bit[31:0] data;
   constraint word_align {addr[1:0] == 2'b0;}
endclass
...
Bus bus = new;
repeat (50) begin
   if ( bus.randomize() == 1 )
   $display ("addr = %16h data = %h\n", bus.addr, bus.data);
   else $display ("Randomization failed.\n");
end
```

Covergroup

- Контроль покрытия
- Позволяет определять подмножества контрольных точек, состояния сигналов в них,
- Можно определить бины, каждому неперечисленному значению соответствует отдельный бин.
- Пример:

```
bit [9:0] v_a;
covergroup cg @(posedge clk);
  coverpoint v_a
  {
  bins a = { [0:63],65 };
  bins b[] = { [127:150],[148:191] };  //пересечение
  bins c[] = { 200,201,202 };
  bins d = { [1000:$] }; //1000-1023
  bins others[] = default;
}
endgroup
```

Основные понятия UVM

UVM

- UVM – библиотека классов, которая предоставляет доступ к возможностям по верификации SystemVerilog.
- Основана на объектно-ориентированных возможностях языка SystemVerilog.
- Реализует методологию тестирования, в которой доступны конструкции, такие, как система отчетов, база данных конфигураций, фабрика для переопределения типов (UVM Factory) и т.д..
- Передачи основаны на TLM (из SystemC)
- Стандарт IEEE 1800.2

Фазы UVM

- Фазы сборки (build)
  - build
  - connect
  - end_of_elaboration
- Фазы исполнения (run)
  - start_of_simulation
  - run*)
    - reset
    - configure
    - main
    - shutdown
- Фазы очистки (clean-up)
  - extract
  - check
  - report
  - final

*) у каждой фазы в run есть pre- и post- фазы

UVM

- Фазы реализуются как методы классов
- Компоненты UVM могут предоставлять методы для каждой фазы, таким образом реализуется управление структурой теста. Можно в рамках тестбенча менять поведение компонент. Заменять методы – например, вводить сбойные пакеты вместо правильных.
- Могут вводиться дополнительные пользовательские фазы

Пример Verilog

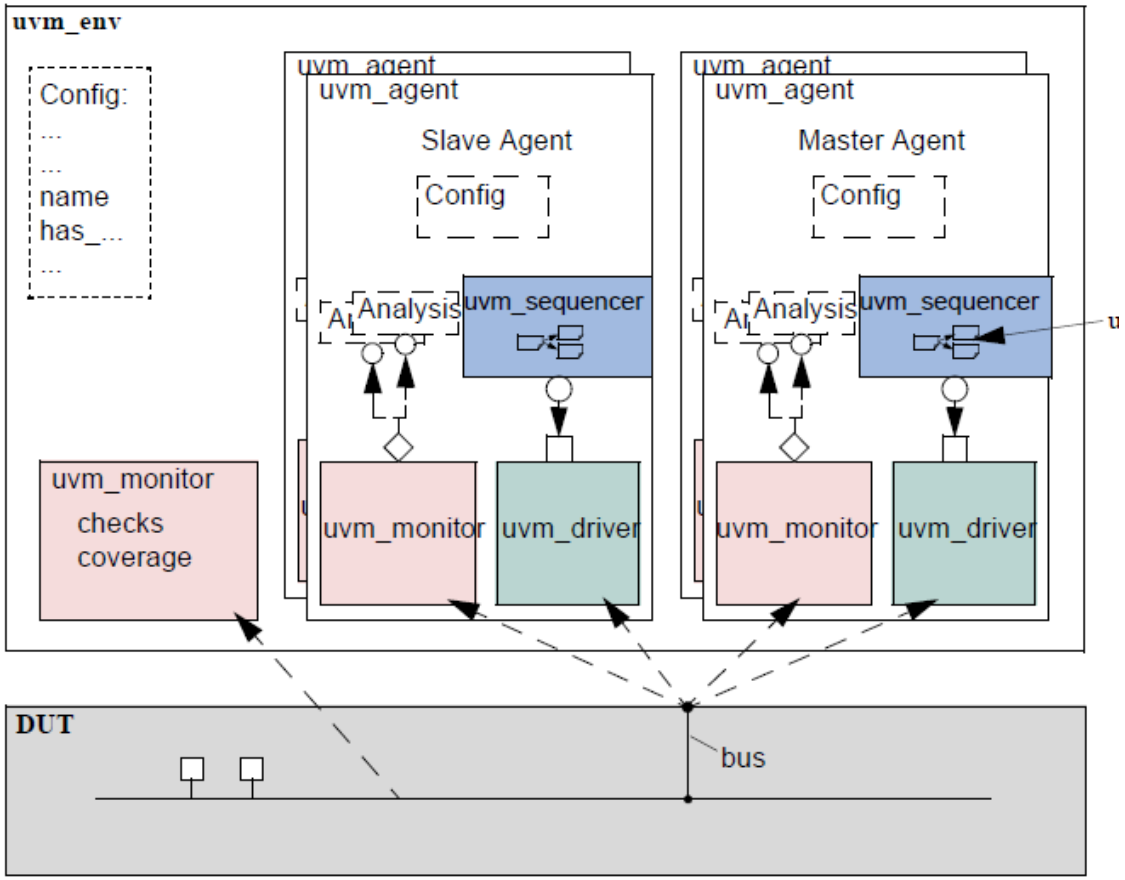
```
module A;
   wire w;
   B B_instance( .p(w) );
   C C_instance( .q(w) );
endmodule
```

Пример UVM

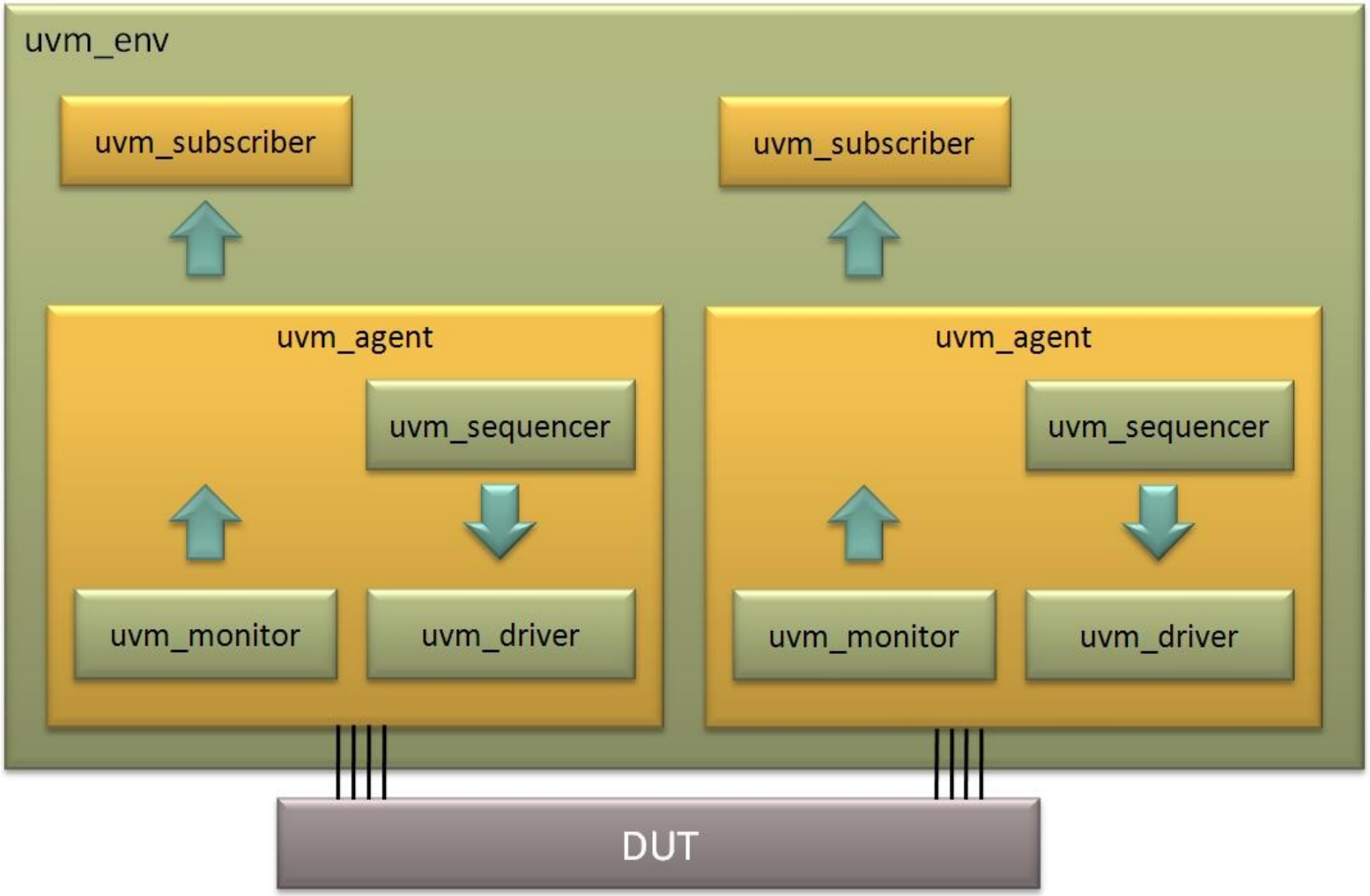
```
class A extends uvm_component;
   `uvm_component_utils(A)
   B B_h;
   C C_h;
   function new(string name, uvm_component parent);
       super.new(name, parent);
   endfunction
   function void build_phase(uvm_phase phase);
       super.build_phase(phase);
       B_h = B::type_id::create("B_h", this);
       C_h = C::type_id::create("C_h", this);
   endfunction
   function void connect_phase(uvm_phase phase);
       B_h.p_port.connect( C_h.q_export );
   endfunction
endclass
```

Основные типы компонент UVM

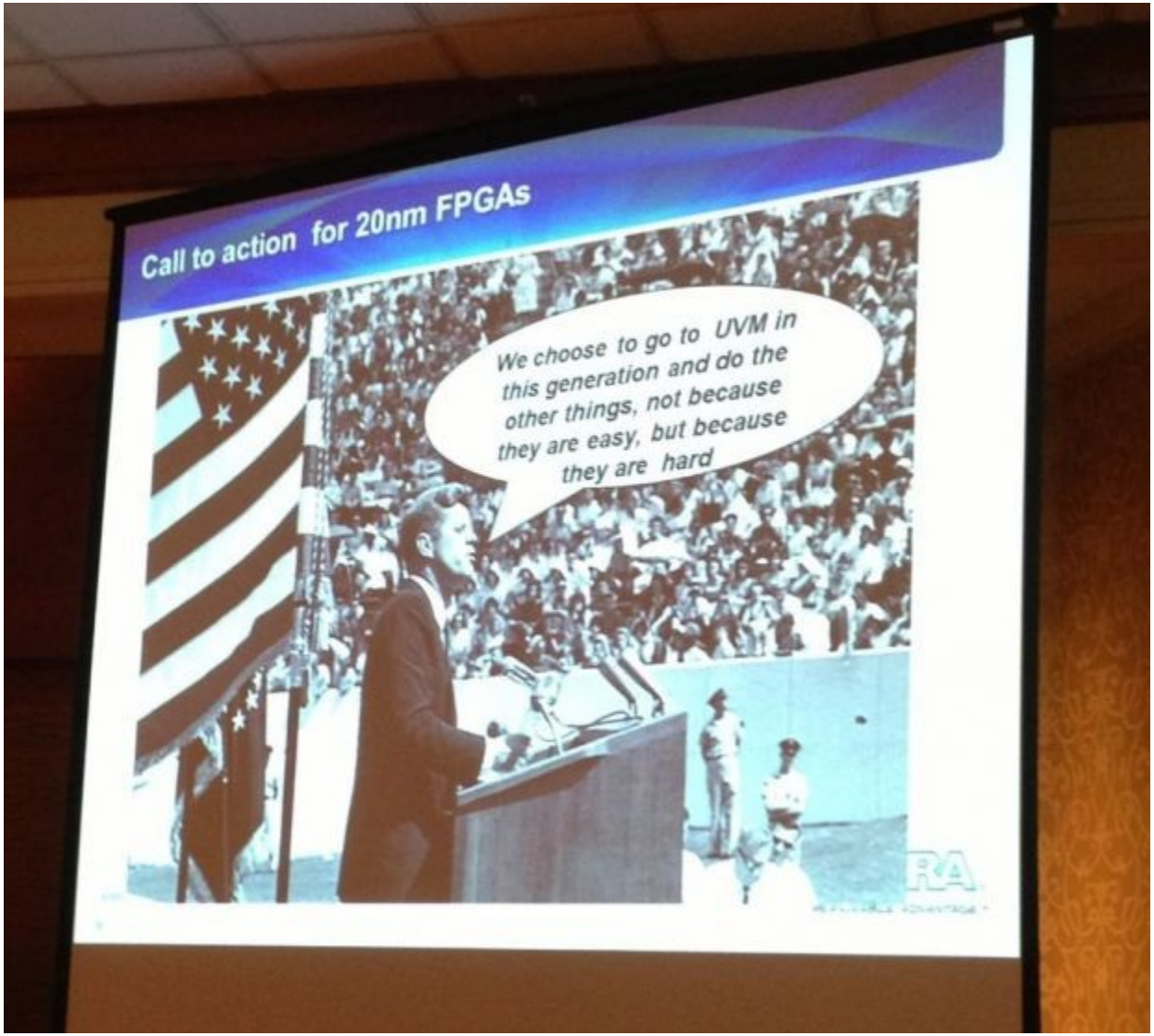
- Основные типы компонент UVM, от которых осуществляется наследование
  - uvm_component
  - uvm_agent
  - uvm_driver
  - uvm_env
  - uvm_monitor
  - uvm_scoreboard
  - uvm_sequencer
  - uvm_subscriber
  - uvm_test



Тестовое окружение UVM



UVM – это не просто



http://ow.ly/i/1Bnq5

Пример UVM

Verilog - модуль для верификации

```
module regfile(clk, reset, addr, cs, wr_n, dwr, drd);

   input clk, reset, cs, wr_n;
   input [1:0] addr;
   input [31:0] dwr;
   output reg [31:0] drd;

   reg [31:0] dreg[3:0];
   integer i;

   always @(posedge clk or negedge reset)
   begin
       if (!reset) for (i=0;i<4;i=i+1) dreg[i]<=0;
       else if (cs & !wr_n) dreg[addr]<=dwr;
   end

   always @(*)
   begin
       drd=dreg[addr];
   end

endmodule
```

Общая структура теста

```
//Включение макросов UVM
`include "uvm_macros.svh"

//Интерфейс тестируемого модуля
interface dut_if;
   …
endinterface

//Создание экземпляра тестируемого модуля на Verilog
module dut(dut_if dif);
   import uvm_pkg::*;
   regfile real_dut(.clk(dif.clock), .reset(dif.reset), … );
endmodule : dut

//Пакет, определяющий все пользовательские классы теста
package my_pkg;
   import uvm_pkg::*;
endpackage: my_pkg

//Модуль верхнего уровня
module top
   import uvm_pkg::*;
   import my_pkg::*;
   …
   run_test("my_test");
   …
endmodule : top
```

Пакет классов теста

```
//Пакет, определяющий все классы в тестовом окружении
package my_pkg;
    import uvm_pkg::*;
    //Класс транзакции my_transaction, формирующий данные (constrained randomisation)
    class my_transaction extends uvm_sequence_item;
    //Класс сиквенсера для контроля последовательности транзакций типа my_transaction
    typedef uvm_sequencer #(my_transaction) my_sequencer;
    //Класс последовательности транзакций типа my_transaction
    class my_sequence extends uvm_sequence #(my_transaction);
    //Класс драйвера, который считывает транзакции из сиквенвера
    //и формирует воздействия на DUT
    class my_driver extends uvm_driver #(my_transaction);
    //Класс окружения, создающий и связывающий сиквенсер и драйвер
    //Прим.: Обычно экземпляры сиквенсера, драйвера, монитора и компоненты контроля
    //создаются и связываются в классе агента, а сам агент уже создается в окружении
    class my_env extends uvm_env;
    //Класс теста, создающий и объединяющий окружение, воздействия
    //и объекты конфигурации теста
    class my_test extends uvm_test;
endpackage: my_pkg
```

Класс транзакции

```
class my_transaction extends uvm_sequence_item;
    //Переменные с псевдослучайными значениями (запись в устройство)
    rand bit dir;
    rand int addr;
    rand int dwr;
    //Переменные (считываемые из устройства)
    int drd;
    //Ограничения рандомизации
    constraint c_addr { addr >= 0; addr < 4; }
    constraint c_data { dwr >= 0; dwr < 256; }
    //Регистрация класса в фабрике и генерация методов для доступа к переменным
    //копирование, сравнение, печать и т.д.
    `uvm_object_utils_begin(my_transaction)
    `uvm_field_int(addr, UVM_ALL_ON)
    `uvm_field_int(dwr, UVM_ALL_ON)
    `uvm_field_int(drd, UVM_ALL_ON)
    `uvm_field_int(dir, UVM_ALL_ON)
    `uvm_object_utils_end
    //Doulos и Mentor рекомендуют заменять эти макросы определенными пользователем
    //обработчиками do_* UVM для повышения производительности
    //http://www.sunburst-design.com/papers/CummingsSNUG2014SV_UVM_Transactions.pdf, с.12

    //Конструктор
    function new (string name = "");
        super.new(name);
    endfunction
endclass: my_transaction
```

Класс последовательности

```
class my_sequence extends uvm_sequence #(my_transaction);
    //Регистрация класса в фабрике для доступа к нему других классов
    `uvm_object_utils(my_sequence)
    //Конструктор
    function new (string name = "");
        super.new(name);
    endfunction
    //Задача последовательности
    task body;
        //Установить objection (возражение) для того, чтобы фаза run не закончилась
        if (starting_phase != null) starting_phase.raise_objection(this);
        //Осуществить 16 транзакций
        repeat(16)
        begin
            //Создание новой транзакции
            req = my_transaction::type_id::create("req");
            //Запуск обработки элемента последовательности
            start_item(req);
            //Рандомизация (после этого транзакцию может считать драйвер)
            if(!req.randomize()) `uvm_error("", "Randomize failed")
            //Завершение обработки
            finish_item(req);
        end
        //Тестирование завершено, снимаем objection
        if (starting_phase != null) starting_phase.drop_objection(this);
    endtask: body
endclass: my_sequence
```

Класс драйвера

```
class my_driver extends uvm_driver #(my_transaction);
    `uvm_component_utils(my_driver)
    //Создание виртуального интерфейса для связи с физическим интерфейсом в DUT
    virtual dut_if dut_vi;
    //Конструктор
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction
    //В фазе build получаем ссылку на интерфейс DUT из базы данных конфигурации
    function void build_phase(uvm_phase phase);
        if( !uvm_config_db #(virtual dut_if)::get(this, "", "dut_if_ref", dut_vi) )
        `uvm_error("", "uvm_config_db::get failed")
    endfunction
    //Задача фазы run
    task run_phase(uvm_phase phase);
        forever
        begin
            //Запрос новой транзакции у сиквенсера
            seq_item_port.get_next_item(req);
            //Формирование передачи на физическом уровне
            @(posedge dut_vi.clock);
            dut_vi.cs  = 1'b1;      dut_vi.wr_n = req.dir;
            dut_vi.addr = req.addr; dut_vi.dwr = req.dwr;
            dut_vi.drd = req.drd;
            //Закончить обработку
            seq_item_port.item_done();
        end
    endtask
endclass: my_driver
```

Класс окружения

```
class my_env extends uvm_env;
    `uvm_component_utils(my_env)
    //Объявление экземпляров сиквенсера и драйвера
    my_sequencer m_seqr;
    my_driver    m_driv;
    //Конструктор
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction
    //Вызов конструктора сиквенсера и драйвера
    function void build_phase(uvm_phase phase);
        m_seqr = my_sequencer::type_id::create("m_seqr", this);
        m_driv = my_driver   ::type_id::create("m_driv", this);
    endfunction
    //Соединение порта драйвера с экспортом сиквенсера
    //Драйвер считывает транзакции из сиквенсера
    function void connect_phase(uvm_phase phase);
        m_driv.seq_item_port.connect( m_seqr.seq_item_export );
    endfunction
endclass: my_env
```

Класс теста

```
class my_test extends uvm_test;
    `uvm_component_utils(my_test)
    //Объявление экземпляра окружения
    my_env m_env;
    //Конструктор
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction
    //Вызов конструктора окружения в фазе сборки (build)
    function void build_phase(uvm_phase phase);
        m_env = my_env::type_id::create("m_env", this);
    endfunction
    //Управление последовательностью в фазе выполнения (run)
    task run_phase(uvm_phase phase);
        //Создание и рандомизация последовательности
        my_sequence seq;
        seq = my_sequence::type_id::create("seq");
        if( !seq.randomize() )
            `uvm_error("", "Randomize failed")
        seq.starting_phase = phase;
        //Запуск последовательности
        //Можно менять параметры, рандомизировать и запускать несколько раз
        seq.start( m_env.m_seqr );
    endtask
endclass: my_test
```

Модуль верхнего уровня

```
module top;
    //Импорт стандартного пакета UVM и созданного нами my_pkg
    import uvm_pkg::*;
    import my_pkg::*;
    dut_if dut_if_inst(); //Экземпляр интерфейса
    dut my_dut(.dif(dut_if_inst)); //Экземпляр DUT
    //Генератор ТИ
    initial
    begin
        dut_if_inst.clock = 0;
        forever #10 dut_if_inst.clock = ~dut_if_inst.clock;
    end
    //Генерация сброса
    //Вариант лучше: http://www.sunburst-design.com/papers/HunterSNUGSV_UVM_Resets_paper.pdf
    initial
    begin
        dut_if_inst.reset = 0;
        #5 dut_if_inst.reset=1;
    end
    //Запуск тестбенча
    initial
    begin
        //Сохранение в базе данных конфигурации информации об интерфейсе DUT
        uvm_config_db #(virtual dut_if)::set(null, "*", "dut_if_ref", dut_if_inst);
        //Запуск теста с использованием библиотеки UVM для класса my_test
        run_test("my_test");
    end
endmodule: top
```

UVM Express и UVM Connect

UVM Express

- Набор методологий, стилей кодирования и применения UVM для упрощения и повышения производительности функциональной верификации.
- Методологии позволяют повысить уровень абстракции тестов с использованием функций и задач BFM, определить функциональное покрытие и псевдослучайную генерацию воздействий с ограничениями.
- UVM Express разработан для того, чтобы упростить внедрение UVM.

https://verificationacademy.com/topics/verification-methodology/uvm-express

UVM или UVM Express?

- Верификацией и проектированием занимаются различные команды?
  - Нет -> UVM Express
  - Да -> полный UVM
- Вы уже используете высокоуровневый язык для верификации?
  - Нет -> UVM Express
  - Да -> полный UVM

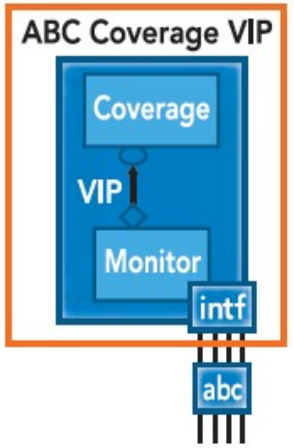
В чем отличие от полного UVM?

- Предполагается, что разработчик реализует тесты с использованием BFM
- UVM Express предоставляет возможность постепенного вхождения в маршрут проектирования с применением UVM.
- На каждом шаге появляются дополнительные возможности.
- Постепенно пользователи добавляют возможности по контролю покрытия и управлению процессом верификации.
- Если покрытие недостаточно, пользователь дорабатывает тестовое окружение, обнаруживая недостаточное покрытие, пользователь использует дополнительные возможности.

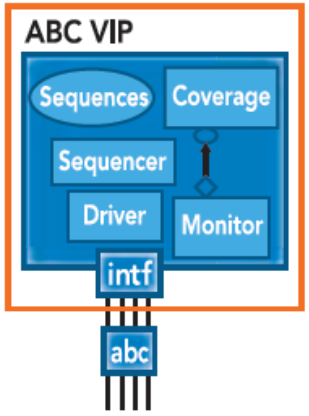
Последовательность

- Первый шаг - BFM
  - желательно также освоение SystemVerilog.

- Второй шаг – функциональное покрытие
  - Освоение использования мониторов и контроля покрытия



- Третий шаг – рандомизация



- Далее переход к полному UVM для управления конфигурацией и программой теста, контроля функционального покрытия и покрытия кода

UVM Connect

- Новая библиотека на основе UVM, предоставляющая коммуникации в соответствии с моделями TLM1 и TLM2, а также API команд между SystemC и моделями UVM и компонентами на SystemVerilog.
- UVM Connect обеспечивает возможность разрабатывать новые интегрированные окружения для верификации, использующие преимущества возможности SystemC и SystemVerilog.

Материалы

- https://verificationacademy.com/

Mentor Graphics Verification Academy

Содержит обучающие материалы по верификации, в том числе UVM Cookbook (обучающие материалы по UVM)

- http://www.accellera.org/home/

Accellera Systems Initiative

Организация, разрабатывающая и поддерживающая стандарты в области проектирования электронных устройств. Членами организации являются ведущие производители полупроводниковых устройств и САПР.

Раздел Downloads содержит документацию и референсную реализацию UVM 2020 и 2017

Материалы

Дополнительные примеры, статьи, видео и обучающие материалы по методологиям верификации

- http://doulos.com
- http://www.sunburst-design.com