Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ **ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ**

КАФЕДРА **КОМПЬЮТЕРНЫЕ СИСТЕМЫ И СЕТИ (ИУ6)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.01 Информатика и вычислительная техника**

Отчет
по лабораторной работе № 2

по дисциплине «Организация ЭВМ и систем»

Тема: «Создание верификационного окружения»

Студент группы ИУ6-73Б _____ В.К. Залыгин
(Подпись, дата) (И.О. Фамилия)

Преподаватель _____ М.П. Калитвенцев
(Подпись, дата) (И.О. Фамилия)

2025 г.

Цель работы

Освоить базовые средства SystemVerilog, необходимые для функциональной верификации, и на их основе разработать каркас верификационного окружения для DUT по индивидуальному варианту. В рамках работы требуется научиться описывать пользовательские типы данных, массивы, процедуры function/task, использовать основные принципы ООП (наследование и полиморфизм), а также подготовить структуру проекта для дальнейшего написания тестов.

Теоретические сведения

SystemVerilog в контексте функциональной верификации используется как сочетание синтезируемого кода (сам DUT и элементы, взаимодействующие с его портами) и несинтезируемого кода (программная часть тестбенча). Несинтезируемое подмножество языка предоставляет привычные программные конструкции: пользовательские типы (enum, struct), массивы, циклы, процедуры function и task, а также объектно-ориентированный подход (классы, наследование, виртуальные методы). Важное различие процедур заключается в том, что function не тратит симуляционное время, тогда как task может содержать задержки и ожидания событий, поэтому task применяется для временных сценариев и взаимодействия с интерфейсами, а function — для вычислений и подготовки данных.

Основой связи тестового окружения с DUT является интерфейс SystemVerilog (interface), который инкапсулирует все входные и выходные сигналы устройства в одной сущности. Верификационные компоненты получают доступ к этому интерфейсу через виртуальный интерфейс (ссылку), что позволяет изнутри классов читать и выставлять значения сигналов, не «зашивая» конкретные имена и иерархию модулей. Для подключения интерфейса к портам DUT создаётся обёртка (harness), внутри которой инстанцируется DUT и выполняется сопоставление сигналов интерфейса с портами устройства, а также задаются параметры DUT согласно варианту.

Тестовое окружение строится по модульному принципу и включает набор агентов интерфейсов. Агент инкапсулирует работу с конкретной частью интерфейса и обычно состоит из драйвера и монитора: драйвер получает транзакции высокого уровня и преобразует их в сигнальные воздействия на входах DUT согласно протоколу, а монитор наблюдает сигналы на интерфейсе и формирует транзакции на основе валидных передач. Для обмена транзакциями между компонентами применяются mailbox-каналы, а запуск компонентов выполняется параллельно в методе run() окружения через fork..join. Окружение организуется как класс env, который выполняет этапы build() (создание и настройка компонентов, установка режима active/passive и передача виртуального интерфейса), connect() (соединение компонентов) и run() (параллельный запуск), что формирует масштабируемый каркас для дальнейшего развития тестов и последовательностей.

Выполнение работы

В задании 1 были созданы пользовательские типы данных (перечисление и структура), проинициализированы переменные этих типов и выполнен вывод их значений, а также выполнено сложение двух 6-битных переменных с выводом результата. Код приведен в листинге 1. Результат запуска задания показан на рисунке 1.

Листинг 1 – Результат выполнения задания 1

```
/*
  1. Объявите новый перечисляемый тип не менее, чем из 5
значений.
  2. Объявите новый тип структуры, содержащей имя, фамилию и
возраст.
  3. Создайте переменные обоих типов, задайте значения и
выведите их на экран.
  4. Создайте две 6-битных переменных, задайте значения и
выведите на экран результат их сложения.
*/
module data_types_task;

  initial begin

    typedef enum {MON=0, TUE=1, WED=2, THU=3, FRI=4} week_e;

    typedef struct {
```

```
      string name;
      string surname;
      int age;
    } person_s;

    person_s person;
    week_e week;
    bit [5:0] bit61 = 6'b000111;
    bit [5:0] bit62 = 6'b101010;
    person.name = "Vyacheslav";
    person.surname = "Zalygin";
    person.age = 21;

    week = MON;

    $display("person name: %s, surname: %s, age: %0d",
person.name, person.surname, person.age);
    $display("week: %0d", week);
    $display("bits1: %1b, bits2: %0b", bit61, bit62);

    $finish;
  end

endmodule : data_types_task
```

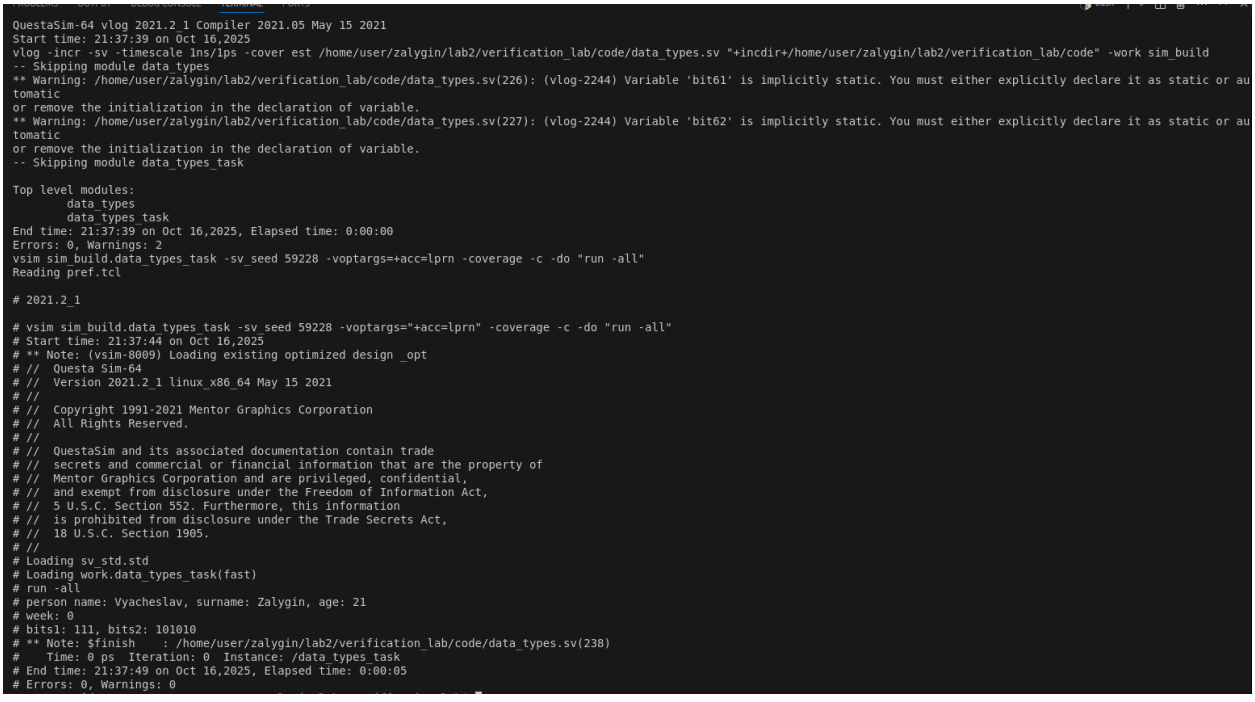

Рисунок 1 – Вывод программы при запуске задания 1

В задании 2 была реализована функция формирования двумерной матрицы заданных размеров с 4-битными элементами: матрица инициализируется нулями, а главная диагональ заполняется значением

4'b1001. Затем матрица была выведена на экран с использованием $display() и $write(). Код приведен в листинге 2. Результат запуска программы показан на рисунке 2.

Листинг 2 – Результат выполнения задания 2

```
/*
  Напишите функцию, которая возвращает двумерную матрицу заданных размеров
  Элементы матрицы - 4-х битные целые числа
  Матрица инициализируется нулями, но главная диагональ должна быть заполнена числами формата 4'b1001
  Выведите матрицу на экран, используя $display() (вывод с переносом на новую строку) и $write() (без)
*/
module arrays_task;

  function void gen_matrix(input int width, input int height, output bit [3:0] matrix [][]);
    bit[3:0] line [];

    // Сначала инициализируется правый индекс
    matrix = new [height];
    foreach (matrix[i]) begin
      // Затем - каждый левый индекс (подмассив)
      line = new [width];
      matrix[i] = line;
      foreach (line[j]) begin
        if (i == j)
          matrix[i][j] = 4'b1001;
      end
    end

  endfunction : gen_matrix

  initial begin

    bit [3:0] matrix [][];
    int width = 20;
    int height = 30;
    bit[3:0] line [];

    gen_matrix(width, height, matrix);

    $display("matrix:");
    foreach (matrix[i]) begin
      line = matrix[i];
      foreach (line[j])
        $write("%4d", matrix[i][j]);
      $write("\n");
    end
```

```
    $finish;
  end

endmodule : arrays_task
```

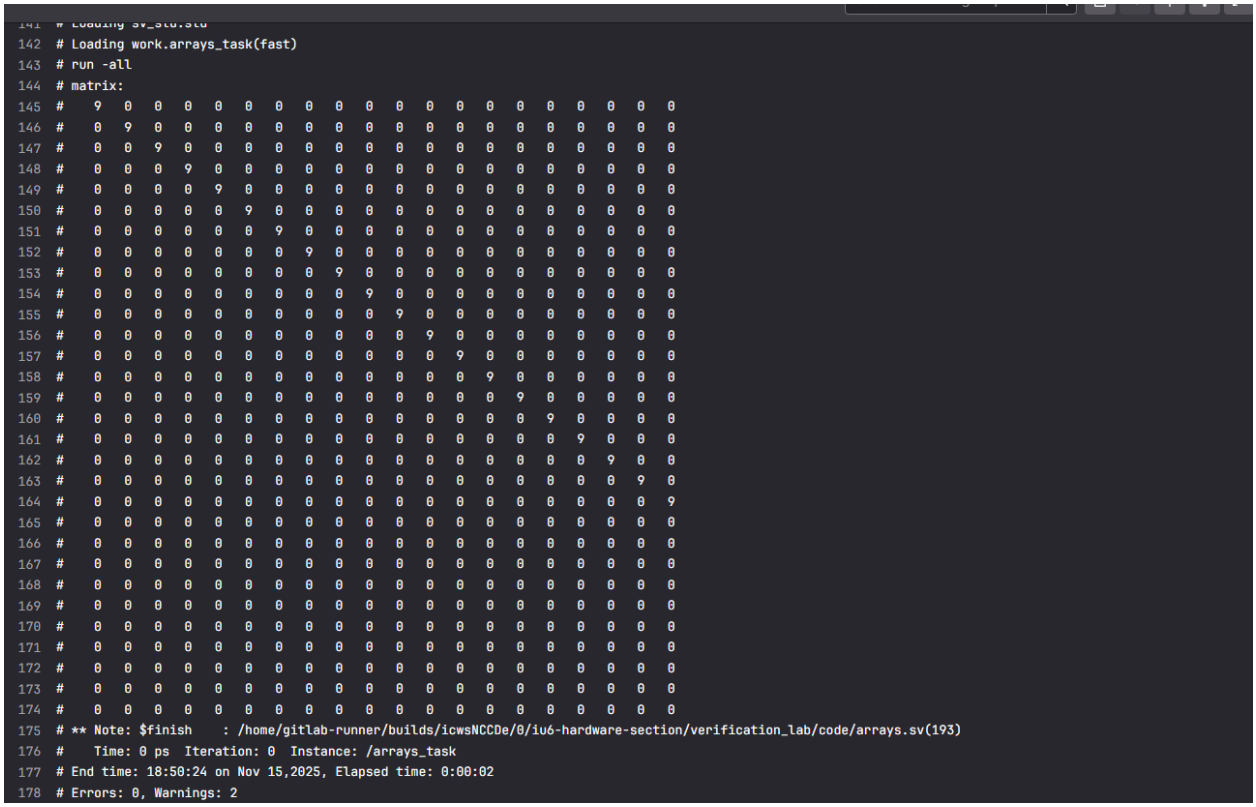

Рисунок 2 – Запуск задания 2

В задании 3 был реализован task, принимающий динамический массив, инициализирующий и заполняющий его через отдельную function с передачей аргумента по ссылке (ref), после чего массив выводился поэлементно с задержкой 5 нс между выводами. Код приведен в листинге 3. Результат запуска программы показан на рисунке 3.

Листинг 3 – Результат выполнения задания 3

```
/*
  Напишите таск, который будет получать на вход динамический
массив, инициализировать его и заполнять значения
  Затем выводить на экран все элементы массива по очереди, через
каждые 5 нс.
  Инициализация и заполнение массива должны быть реализованы
функцией с передачей аргумента по ссылке
  Функция должна вызываться из тела таска
*/
module functions_and_tasks_task;
```

```
  function automatic init_array(ref int arr [][]);
    int line [];

    arr = new [3];
    foreach(arr[i]) begin
      line = new [3];
      foreach(line[j]) begin
        line[j] = j + i;
      end
      arr[i] = line;
    end
  endfunction

  task main_task(int arr [][]);
    int line [];

    init_array(arr);

    $display("arr:");
    foreach (arr[i]) begin
      line = arr[i];
      foreach (line[j]) begin
        $write("%4d", arr[i][j]);
        #5ns;
      end
      $write("\n");
    end
  endtask

  initial begin
    int arr [][];

    main_task(arr);
  end

endmodule : functions_and_tasks_task
```

```
133  # Loading work.functions_and_tasks_sv_unit(fast)
134  # Loading work.functions_and_tasks_task(fast)
135  # run -all
136  # arr:
137  #    0   1   2
138  #    1   2   3
139  #    2   3   4
140  VSIM 2> #
141  # <EOF>
142  # End time: 20:11:59 on Nov 15,2025, Elapsed time: 0:00:02
143  # Errors: 0, Warnings: 2
144  Cleaning up project directory and file based variables
145  Job succeeded
```

Рисунок 3 – Запуск задания 3

В задании 4 были выполнены эксперименты с наследованием классов: создана цепочка наследования и несколько классов-потомков, проанализировано поведение унаследованных методов и ограничения языка при изменении полей в дочерних классах. Код приведен в листинге 4. Результат запуска программы показан на рисунке 4.

Листинг 4 – Результат выполнения задания 4

```
/*
  Поэкспериментируйте с наследованием классов. Создайте
несколько классов, отнаследованных от одного и того же родителя,
  напишите цепочку наследования, где класс C наследуется от B, а
тот наследуется от A, и т.д. и т.п. Понаблюдайте, как ведут себя
  наследуемые методы, как компилятор относится к изменению типа
поля в дочернем классе и т.п.
*/
class A;
  virtual function string get_name_v();
    return "class A";
  endfunction : get_name_v

  function string get_name();
    return "class A";
  endfunction : get_name

  function void print();
    $display("%s", get_name());
    $display("%s", get_name_v());
  endfunction : print
endclass : A

class B extends A;
  function string get_name_v();
    return "class B";
  endfunction : get_name_v

  function string get_name();
    return "class B";
  endfunction : get_name
endclass : B

class C extends B;
  function string get_name_v();
    return "class C";
  endfunction : get_name_v

  function string get_name();
    return "class C";
  endfunction : get_name
endclass : C
```

```
module classes_task;

  initial begin
    A a = new();
    B b = new();
    C c = new();

    a.print();
    b.print();
    c.print();
  end

endmodule : classes_task
```

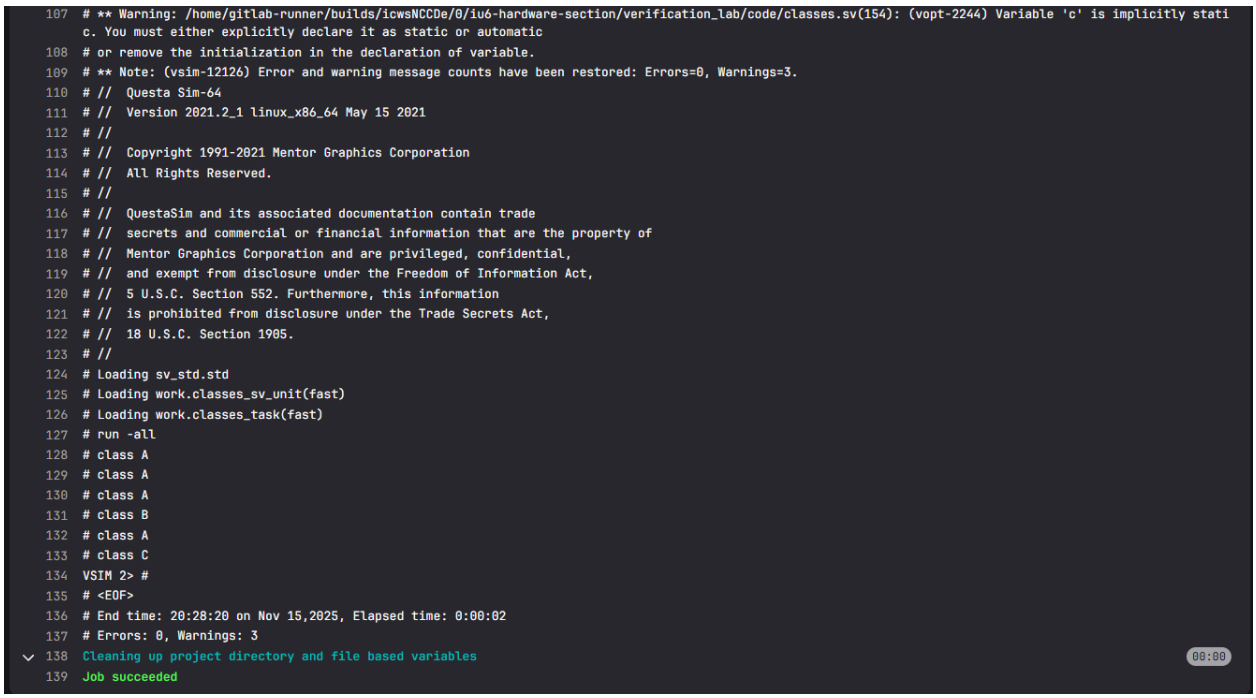

Рисунок 4 – Запуск задания 4

В задании 5 были реализованы родительский и дочерний классы с виртуальным методом, а также функция, принимающая объект родительского класса и вызывающая виртуальный метод; продемонстрирован полиморфизм при передаче в функцию объектов базового и производного классов. Код приведен в листинге 5. Результат запуска программы показан на рисунке 5.

Листинг 5 – Результат выполнения задания 5

```
/*
  Напишите некий простой родительский класс с небольшим набором численных полей и виртуальным методом,
  проводящим операцию над этими полями (сложение, перемножение, сумма массива - не важно) и выдающим результат.
```

```
  Затем напишите дочерний класс, отнаследованный от
вышеуказанного. Переопределите виртуальный метод - пусть он
делает что-нибудь другое,
  но выдает результат того же типа данных (а иначе не получится).
  Вне этих классов напишите функцию, входным аргументом которой
будет объект родительского класса. Внутри функции должен
вызываться
  виртуальный метод родительского класса.
  В теле основной программы вызовите эту функцию и передайте ей
в качестве аргумента спервa объект родительского, затем дочернего
класса.
  Посмотрите, что будет.
*/
/* Your classes here */
class EntryC #(type K, type V, string TEMPLATE);
  K k;
  V v;

  function void make(K key, V value);
    k = key;
    v = value;
  endfunction : make;

  function void print();
    $display(TEMPLATE, k, v);
  endfunction

endclass : EntryC

class AbstractTableC #(type K, type V, string TEMPLATE);

  virtual function void set(K key, V value);
  endfunction : set

  virtual function V get(K key);
  endfunction : get

  virtual function void get_entries(ref EntryC #(K, V, TEMPLATE)
entries []);
    entries = new [0];
  endfunction : get_entries

  function void print();
    EntryC #(K, V, TEMPLATE) entries [];
    get_entries(entries);

    foreach (entries[i])
      entries[i].print();
  endfunction : print

endclass : AbstractTableC
```

```
class HashTableC #(type K, type V, string TEMPLATE) extends
AbstractTableC #(K, V, TEMPLATE);

  protected V mem[K];

  function void set(K key, V value);
    mem[key] = value;
  endfunction : set

  function V get(K key);
    return mem[key];
  endfunction : get

  function void get_entries(ref EntryC #(K, V, TEMPLATE) entries
[]);
    int i = 0;
    EntryC #(K, V, TEMPLATE) entry;
    entries = new [mem.size];
    foreach(mem[key]) begin
      entry = new();
      entry.make(key, mem[key]);
      entries[i++] = entry;
    end
  endfunction : get_entries

endclass : HashTableC


module polymorphism_task;

  initial begin
    HashTableC #(int, string, "entry[%d: %s]") hash_table1 = new
();
    HashTableC #(string, int, "entry[%s: %d]") hash_table2 = new
();
    hash_table1.set(1, "one");
    hash_table1.set(2, "two");
    hash_table2.set("one", 1);
    hash_table2.set("two", 2);

    hash_table1.print();
    hash_table2.print();
  end
endmodule : polymorphism_task
```

В задании 6 по спецификации варианта был разработан интерфейс env_if.sv, включающий все входные и выходные сигналы DUT и предназначенный для использования как виртуальный интерфейс внутри

окружения. Код приведен в листинге 6. Результат запуска программы показан на рисунке 6.

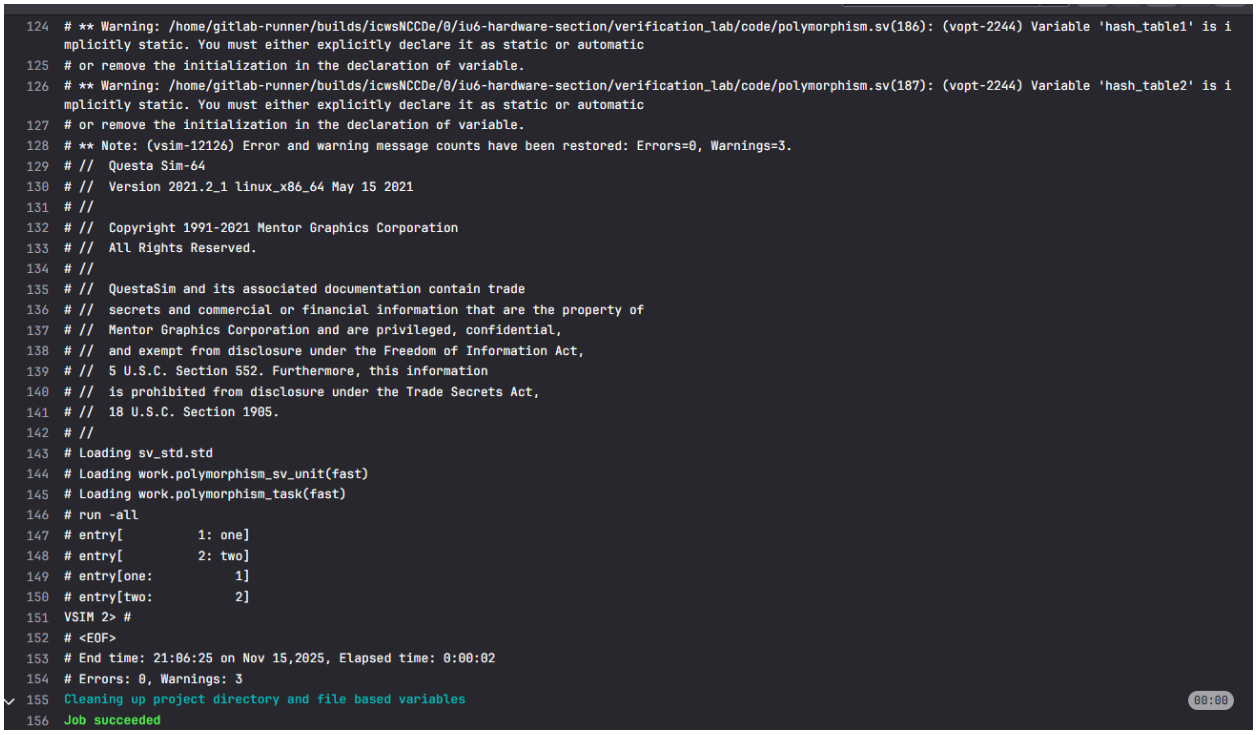

Рисунок 5 – Запуск задания 5

Листинг 6 – Результат выполнения задания 6

```
//////////////////////////////////////////////////////////////////////
////////////////////
// Company: Bauman Moscow State Technical University
// Engineer:
//
// Create date: 06/02/2025
// Project Name:
// Module Name:
// Target Devices:
// Tool Versions:
// Description: DUT interface.
//////////////////////////////////////////////////////////////////////
////////////////////

interface env_if ();
  logic clk;
  logic resetn;

  /*
  Write the rest of the signals and (or) interfaces here
  */
  logic [3:0] in_data;
  logic       in_valid;
  logic [2:0] out_data;
endinterface : env_if
```

В задании 7 была создана обёртка DUT (harness) в файле tb_dut_wrapper.sv: инстанцирован модуль устройства по варианту, заданы требуемые параметры и выполнено подключение всех сигналов интерфейса к портам DUT; также в Makefile обновлено имя модуля DUT. Код tb_dut_wrapper.sv приведен в листинге 7.

Листинг 7 – Результат выполнения задания 7

```
//////////////////////////////////////////////////////////////////////
////////////////////
// Company: Bauman Moscow State Technical University
// Engineer:
//
// Create date: 06/02/2025
// Project Name:
// Module Name:
// Target Devices:
// Tool Versions:
// Description: DUT interface.
//////////////////////////////////////////////////////////////////////
////////////////////

module tb_dut_wrapper (env_if env_intf);

  // Clock signal; DUT input
  logic clk;
  assign clk = env_intf.clk;

  // Reset signal; DUT input
  logic resetn;
  assign resetn = env_intf.resetn;

  /*
  Write the rest of the signal assignments here
  */

  // Important: when assigning DUT inputs, do: assign signal =
intf.signal
  // when assigning DUT outputs, do: assign intf.signal = signal

  logic in_valid;
  assign in_valid = env_intf.in_valid;
  logic[3:0] in_data;
  assign in_data = env_intf.in_data;

  logic [2:0] out_data;
  assign env_intf.out_data = out_data;
```

```
  // DUT module instance
  /* Your DUT instance here */

  mealy_machine_5state_var6_wrap # (
    .INPUT_WIDTH(4)
  ) u_mealy_machine_5state_var7_wrap (
    .clk(clk),
    .resetn(resetn),
    .in_data(in_data),
    .in_valid(in_valid),
    .out_data(out_data)
  );

endmodule : tb_dut_wrapper
```

В задании 8 был определён набор необходимых агентов окружения (помимо готовых агентов clk и reset): выделены агенты для логически разных частей интерфейса, а для одинаковых по протоколу входа/выхода предусмотрено использование двух экземпляров одного агента в активном и пассивном режимах.

Для приоритетного шифратора логично разделять окружение на входной и выходной агенты, потому что это разные по направлению и смыслу части интерфейса: входной агент формирует воздействие (подаёт комбинации входных сигналов на DUT), а выходной агент наблюдает реакцию устройства (снимает результат кодирования и признаки валидности/ошибок, если они предусмотрены). Такое разделение упрощает архитектуру тестбенча: входной агент работает в активном режиме (драйвер + монитор, чтобы фиксировать фактически поданные значения), а выходной — в пассивном режиме (обычно монитор, чтобы собирать реакции DUT независимо от драйвера). Агенты clk_agent и resetn_agent используются как служебные и обеспечивают генерацию тактового сигнала и сброса, поэтому отдельно выделять их в варианте устройства не требуется - они стандартные для любого DUT.

В задании 9 были разработаны транзакции и драйверы для всех необходимых агентов: транзакции унаследованы от base_transaction с переопределением compare(), copy(), print(), а драйверы унаследованы от

env_base_driver с реализацией init_signals(), drive_tran() и (при необходимости) drive_passive().

В задании 10 были разработаны мониторы для всех агентов на основе env_base_monitor: реализованы методы check_validity(), monitor_tran() и (при необходимости) reset(), а собранные транзакции передаются во внешний мир через mon2out_mbx.

Итоговые папки агентов приведены на рисунке 6.

В задании 11 были собраны полноценные агенты путём интеграции транзакции, драйвера, монитора и секвенсора в классы-наследники env_base_agent, а все файлы агентов подключены в env_pkg.sv.

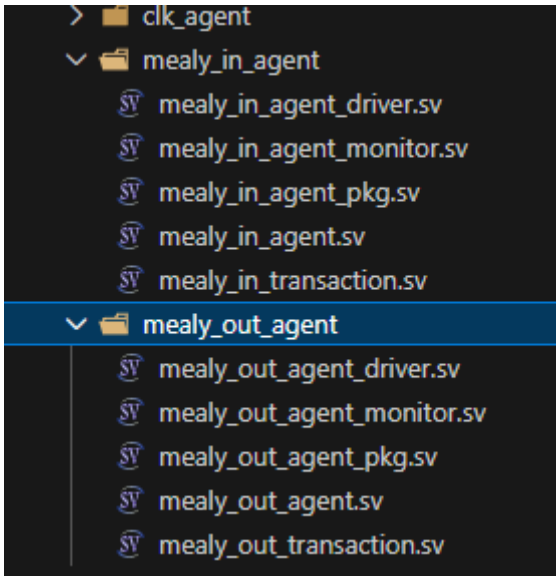

Рисунок 6 - Агенты

В задании 12 написанные агенты были интегрированы в окружение env.sv: добавлены поля агентов, выполнена инициализация и настройка (active/passive), передан stop_event, установлен виртуальный интерфейс через set_vif(), и агенты запущены параллельно в run() через fork..join.

В задании 13 итоговый код был загружен в удалённый репозиторий в личную ветку с запуском CI через пометку [ci lab1], после чего подтверждена успешная сборка и работоспособность окружения по результатам пайплайна.

Вывод

В ходе лабораторной работы были освоены базовые средства SystemVerilog, применяемые при построении тестбенчей: работа с

пользовательскими типами данных и массивами, использование function/task и принципов ООП (наследование и полиморфизм). По индивидуальному варианту для приоритетного шифратора разработаны интерфейс env_if и обёртка DUT, определён и реализован набор агентов (входной и выходной, а также служебные clk/reset), созданы соответствующие транзакции, драйверы и мониторы, после чего агенты были интегрированы в окружение и успешно собраны в единую структуру проекта. В результате получен работоспособный каркас верификационного окружения, готовый для дальнейшего расширения последовательностями и тестами.