Описания соединений мо дулей и КОНЦЕПЦИЯ ИНТЕРФЕЙСА

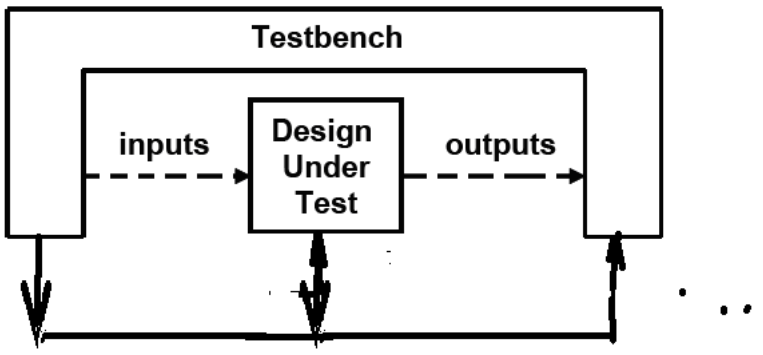
В Verilog использовалось соединение сигналов по позиции или по имени.
SystemVerilog предлагает три упрощенные формы описания связей портов:
1) .name ("dot-name") соединение портов;
2) .* ("dot-star") соединение портов;
3) с помощью интерфейсов.
SystemVerilog выполняет соединение портов с совпадающими по имени сигналами,
упрощая выражение Verilog
.data(data) (листинг а)
до .data в SystemVerilog (листинг б).
Соединение .* (листинг 18.5, в) связывает все порты с соответствующими по имени сигналами.
Примеры соединений портов

а) //Verilog стиль
// Копия модуля с именным соединением портов
```
pc_stack pes (
        .program_counter(program_counter),
        .program_address(program_address),
        .clk(clk),
        .resetN (resetN),
        .instruct_reg (vr_s_ruct_reg),
        .data_bus (data_bus),
        .status_reg(status_reg));

prom prom (.dout(program_data), .clk(clk),.address(program_address));
```

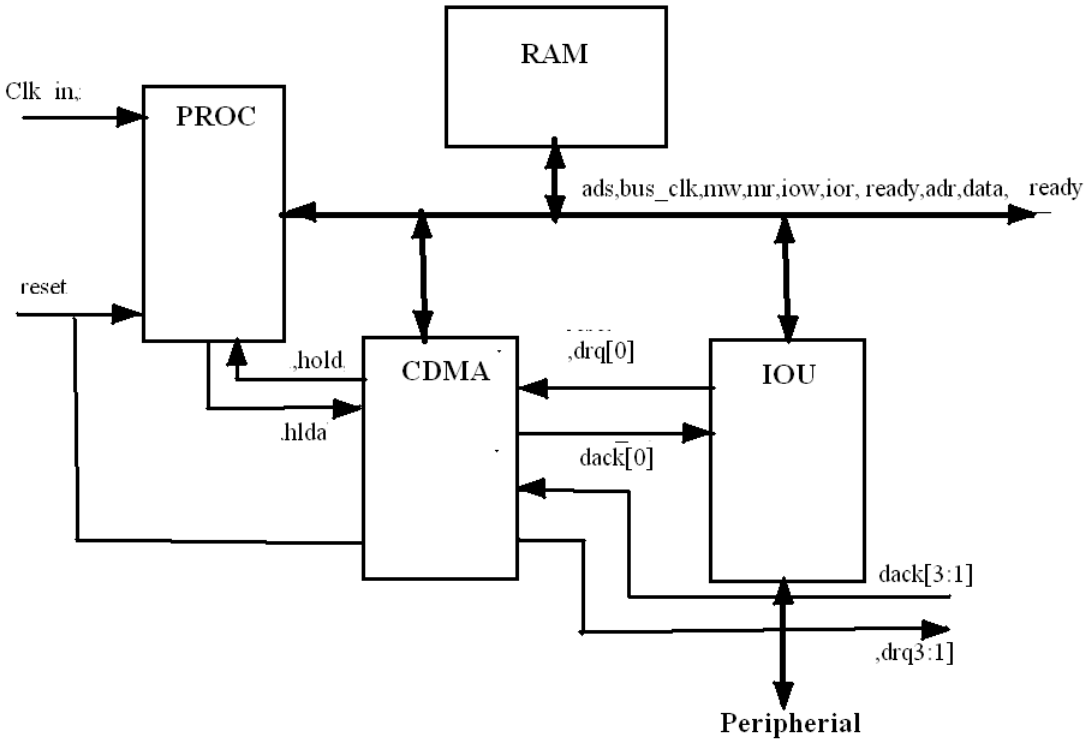
б) // SystemVerilog стиль с использованием .name
(освобождает от правила послеовательнлого перечисления, но я не рекомендую)
// Копия модуля с соединением портов .name
```
pc_stack pes (.program_counter, .program_address,
                clk, resetN, instruct_reg, data_bus, ,status_reg);

prom prom ( .dout(program_data), .clk(,.address(program_address));
```

в) SystemVerilog стиль с использованием *
// Копия модуля с соединением портов .*
```
pc_stack pes (.*,.instruct_reg (vr_s_ruct_reg));
prom prom (.*, ,dout(program_data),
.address(program_address));
```

Концепция интерфейса обеспечивает компактную обозримую запись и эффективный reuse как на этапе проектирования, так и в программах тестированияыя



Концепцию интерфейса рассмотрим на примере фрагмента микропроцессорнрй стстемы, содержащей прпоцессор, память, устройство ввода-вывода и контроллер прямого доступа к памяти



"classical " verilog style

```
Module processor (ads,bus_clk,mw,mr,iow,ior, ready,adr,data, ready,
        Clk_in,reset,hold,hlda);
 Output reg ads,bus_clk,mw,mr,iow,ior,hlda;
Input wire  Clk_in,reset,hold,ready;
Inout wire  [31:0] data
Output reg [31:0] adr ;
< function description>
    Endmodule;

Module ram (ads,bus_clk,mw,mr ,adr,data, ready);
Parameter my_adr=b0;
Input wire  ads,bus_clk,mw,mr, ready;
Output reg ready
Input  reg [31:0] adr;
output reg ready;
inout wire [31:0] data;
< function description>
 // if ads&& adr==my_adr…) ..data=….
  Endmodule

Module IOU (ads,bus_clk iow,ior, ready,adr,data, dack, drq);
Parameter my_adr=0;
Input wire ads,bus_clk, iow,ior
Output reg ready;
Input wire  [31:0] adr;
output reg ready;
inout wire [31:0]  data;
input wire dack;
output wire  drq;
< function description>
```

```
// if ads&& adr==my_adr…) ..data=….
 Endmodule

Module cdma((ads,bus_clk,mw,mr,iow,ior, ready,adr,data,
        ,reset,hold,hlda);
Parameter my_adr=0;
Inout wire  ads,bus_clk, iow, ioriow,ior, ready;
Output  wire;mw,mr,
Input wire reset,hlda;
Output reg hold
Inout wire  [31:0] adr,data;
Input wire  [3:0] drq;
Output reg [3:0] dack;
< function description>
    Endmodule

Modul top (Clk_in,reset,)
wire ads,bus_clk,mw,mr,iow,ior, ready;
wire  hold,hlda;
wire [31:0] adr,data;
wire [3:0] drq,dack;
processor v1(ads,bus_clk,mw,mr,iow,ior, ready,adr,data, ready,
        Clk_in,reset,hold,hlda);
Ram # ’d20 …….
        v2 (ads,bus_clk,mw,mr ,adr,data, ready);
iou # ’d18……
       v3(ads,bus_clk ,iow,ior, ready,adr,data….);
cdma   v4 (ads, bus_clk,mw,mr,iow,ior, ready,adr,data,
        ,reset,hold,hlda,…..);
endmodole;
```

Для больших проектов запись такого способа соединений становится очень
Сложной, а главное плох-читаемой и плохо защищенной от ошибок

К недостаткам использоания традиционного описания портов в больших Verilog проектах можно отнести:
1) декларации дублируются в нескольких модулях;
2) коммуникационные протоколы дублируются в нескольких модулях, что
повышает возможность возникновения ошибки в декларациях различных модулей;
3) изменение в спецификации проекта требует выполнения модификации в
нескольких модулях.

SystemVerilog предлагает новую мощную конструкцию для описания портов
- интерфейс. Интерфейс позволяет группировать сигналы вместе, при этом
каждый модуль, использующий сигналы из интерфейса, рассматривает его
как единый порт
Интерфейс позволяет: дискретные сигналы, коммуникационный протокол и порты определять в одном месте; выполнять проверку и верификацию подпрограмм прямо в интерфейсе. Интерфейс может содержать декларацию типов, задач, функций, процедурных блоков, программных блоков
и ассерций. для каждого модуля, подключаемого с помощью интерфейса,
сигнал может быть входом, выходом и двунаправленным портом.
Предпочтительнее использовать тип logic для описания сигналов в интерфейсе,
это позволит передавать им значения с помощью процедурных операторов.

Основные отличия между интерфейсом и модулем:
1) интерфейс не может содержать описание иерархии проекта;
2) нельзя использовать модуль в списке портов;
3) интерфейс может включать конструкцию modport.

SYSTEM_VERILOG STYLE WITH INTERFACE

```
Interface sys_bus  (ads,bus_clk,mw,mr,iow,ior, ready,adr,data, ready);// линии интефейса
Paramwter  adr_len=32;
Parameter data_len=32;
Logic ads,bus_clk,mw,mr,iow,ior, ready,adr,data, ready; // спецификация линий
Logic [adr_len-1:0] adr;
Logic [data_len-1:0] data;
Endinterface;

Module  processor (sys_bus proc, // линии интефейса
Clk_in,reset,hold,hlda);//прочие связи;
 Input logic  Clk_in,reset,hold; // спецификация не интерфейсных линий
 Output  logic ,hlda;
< function description>
 always_ff  @.....
         proc.ads =
     Endmodule;

Module ram (sys_bus bbb);
Parameter my_adr=0;
< function description>
 // if bbb.ads&& bbb.adr==my_adr…) ..bbb.data=….
  Endmodule

Module IOU (sys_bus iii,dack,drq , <порты связи с периферией>);
Parameter my_adr=0;
//logic ads,bus_clk, iow,ior, ready;
input logic dack;
output logic drq;
<спецификация соединения с периферией>
< function description>
 // if iii.ads&& iii.adr==my_adr…) ..iii.data=….
  Endmodule

Module cdma(sys_bus dm,
 ,reset,hold,hlda,drq,dack);
Parameter my_adr=0;
Input logic Clk_in,reset,hlda;
Output logic hold;
Output  logic  [3:0] dack;
 Input logic [3:0] drq;

< function description>
     Endmodule

Module top (Clk_in,reset,)
Logic [32] adr,data;
logic ads,bus_clk,mw,mr,iow,ior, ready;
logic  hold,hlda;
logic [3:0] drq,dack; ,,,,
sys_bus bus  ();//присваивание линиям интерфейса имени
processor v1(.proc(bus),
          .clk_in,.reset,.hold(hold),.hlda(hlda));
Ram # ’20 b<БАЗОВЫЙ АДРЕС>…….
         v2 (.bbb(bus));//. . . . . .
 IOU  # 16’b <БАЗОВЫЙ АДРЕС>……
       v3(.iii(bus),dack(dack[0], drq(drq[0]), ..<соединения с периферией>;
cdma  # 16’b <БАЗОВЫЙ АДРЕС> v4 ( .dm(bus),hold(hold),hlda(hlda),);
endmodule;
```

Interface syntax

interface_declaration ::=
interface_ansi_header [timeunits_declaration] { non_port_interface_item }
endinterface [**:** interface_identifier]
| { attribute_instance } **interface** interface_identifier **(.*) ;**
[timeunits_declaration] { interface_item }
endinterface [**:** interface_identifier]
| **extern** interface_ansi_header
interface_ansi_header ::=
{attribute_instance } **interface** [lifetime] interface_identifier
[parameter_port_list] [list_of_port_declarations] **;**
modport_declaration ::= **modport** modport_item { **,** modport_item } **;**
modport_item ::= modport_identifier **(** modport_ports_declaration { **,** modport_ports_declaration } **)**
modport_ports_declaration ::=
{ attribute_instance } modport_simple_ports_declaration
| { attribute_instance } modport_tf_ports_declaration
| { attribute_instance } modport_clocking_declaration
modport_clocking_declaration ::= **clocking** clocking_identifier
modport_simple_ports_declaration ::=
port_direction modport_simple_port { **,** modport_simple_port }
modport_simple_port ::=
port_identifier
| **.** port_identifier **(** [expression] **)**
modport_tf_ports_declaration ::=
import_export modport_tf_port { **,** modport_tf_port }
modport_tf_port ::=
method_prototype
| tf_identifier
import_export ::= **import** | **export**
interface_instantiation ::=
interface_identifier [parameter_value_assignment]
hierarchical_instance { **,** hierarchical_instance } **;**

Port declaratiom
inout_declaration ::=
inout net_port_type list_of_port_identifiers
input_declaration ::=
input net_port_type list_of_port_identifiers
| **input** variable_port_type list_of_variable_identifiers
output_declaration ::=
output net_port_type list_of_port_identifiers
| **output** variable_port_type list_of_variable_port_identifiers
interface_port_declaration ::=
interface_identifier list_of_interface_identifiers
| interface_identifier **.** modport_identifier list_of_interface_identifiers

Упрощенное соединение портов SystemVerilog с помощью стилей .name и .*
может быть использовано и для интерфейса. Пример с листинга 19.4 может
иметь более компактный вид, благодаря комбинации использования интерфейсов
и соединения портов Л

Листинг . Упрощение соединения портов

```
// Декларация интерфейса
interface main_bus (input logic clock, resetN, test_mode);
wire [15:0] data;
wire [15:0] address;
logic I 7:0] slavejnstruction;
logic slave_request;
logic bus_grant;
```

```
logic bus_request;
logic slave_ready;
logic data_ready;
logic mem_read;
logic mem_write;
endinterface

// Модуль верхнего уровня

module top (input logic clock, resetN, test_mode);
logic hold,hlda ,drq,dack …..

logic [ 7:0} instruction, nextinstruction, data_b;
main_bus bus (.* );
processor vl (.*,……) ;
ioi  # 16'b <ИНДИВИДУАЛЬНЫЙ  АДРЕС> v3 (.*,……) ;
ram  # 20'b <БАЗОВЫЙ АДРЕС> v2 (.*);
cdma  16'b <ИНДИВИДУАЛЬНЫЙ  АДРЕС> v4 ( .dm(bus),hold(hold),hlda(hlda),);
endmodule
```

Установка режима порта
В предыдущих примерах применялось соединение без указания направления сигнала в интерфейсе. Однако один и тот же сигнал в различных модулях может играть различную роль. Например, для одного он будет входом, а для другого выходом. В этом случае такой порт внутри интерфейса объявляется как modport.

Конструкция modport позволяет группировать сигналы и описывать их направление, что дает возможность выполнять дополнительную проверку при передаче информации и исключать связанные с этим ошибки.
Modport определяет направление порта, которое соответствует модулю. Интерфейс может содержать любое число деклараций modport, каждая описывает один или несколько модулей, рассматривая сигналы в интерфейсе.

```
interface bus ();
//состав цепей интерфейса
logic check_point;
logic [2:0] data_in;
logic [2:0] data_out;

//режимы интерфейса для тестирующих
// блоков
modport stim
(output data_in, data_out, check_point);
modport monit
(input data_in, data_out, check_point);
//так может выглядеть режим интерфейса
//для испытуемого устройства
 modport syst
(input data_in, output data_out);

endinterface
module complex(bus.syst  mb);
wire [7:0] y;   // внутренние связи модуля
decod # // !– Задать параметры вхождения
 mod1(.x_in(mb.data_in),.y_out(y));
or mod2(mb.data_out, y[0],y[5],y[7]);
or   mod…..  //
or   mod4 …../;
endmodule
```

```
module mot (  bus.monit uuu);
 parameter truth_table =8 'b 10001001;
   logic vt;
   logic error;
  assign  vt =truth_table[uuu.data_in] ;
  always @ (posedge uuu.check_point)
     error = uuu.data_out[0]== vt ?  0:1;
endmodule

program stimul (bus.stim uuu);
      parameter delay = 6;
     integer j;
      initial begin
       uuu.check_point=0;
        for (j=0; j<=7;j=j+1) begin
       #10 ;
          uuu.data_in=j;
        uuu.check_point= # delay 1;
           #3 ;
          uuu.check_point=0;
                                end
                    end
endprogram
```

```
module test_sv;
   logic [2:0] data_in;
   logic [2:0] data_out;
   logic chech_point;
  stimul test_sequence  ();
   bus  uuu (); //   оператор вхождения блока типа  bus, uuu- имя входения
                 //  список соответствий здесь не нужен потому, что имена фактических и
                 //формальных параметров совпадают
  mot  m0 (.uuu(uuu));
   complex
              m1 ( .mb (uuu))  // либо так либо как ниже
   //m1(uuu.data_in,uuu. data_out[0],uuu.data_out[1],uuu.data_out[2]);
endmodule
```

Если направления портов не описаны, то по умолчанию все порты интерфейса имеют режим inout и тип ref. Кроме этого, в интерфейсе можно объявлять внутренние сигналы, являющиеся локальными для него.

Интерфейсы могут использовать параметры для настройки размеров, которые могут меняться при создании копии интерфейса

Интерфейс может содержать процедурные блоки (always, always_comb, always_
ff, aiways_latch, initial или final) и операторы непрерывного присвоения
(assign).
Это свойствр может быть использовано для контроля протоколов, управления форматами передаваемых данных даже для буферизации.

```
interface bus ( ) ;
parameter d_len=32;
parameter a_len=32;
parameter crc_len=8;
 logic adr[a_len-1:0];
logic data [d_len-1:0];
logic crc [crc_len-1:0];
logic  ads,clk, reset,ready,wr,rd,error;
logic error1,error2;

function calc_crc();
endfunction;

void task timing_controle();
enum {idle,prep,wrk} state;
logic cycle_running,active;
 begin
 @( posedge clk or posedge reset);
    if (!reset)
begin
 state=idle; error1=1'b0; cycle_running='b0;
  end
   else
   case (state)
      idle:  if  (! cycle_running &&  (!wr && !rd) )
                 //  syncro error
                 error1=1'b1;
            else if (!ads)   begin state=prep; cycle_running='b1;//
                             end
      prep:   if (!rd|| !wr)  state  = wrk;
      wrk: if (!ready) begin  state=idle; cycle_running='b0;
                       end

     endcase;
  end
endtask;
    // assert calc_src/=src   $write ( "crc error");
 always  @( posedge rd or posedge wr)
       error2= calc_crc() !=crc;
 assign error=error1 || error2;
timing_controle();

endinterface
```

Декларация подпрограммы может располагаться в секции MODPORT
. Подпрограммы могут импортироваться из других программных модулей
Простейшая форма импортирования задачи
или функции просто описывает ее имя. Синтаксис:
modport (import <task_function_name>);
Пример использования:

```
modport in (import Read,
import parity_gen,
input clock, resetN );
```

Параметризированные типы данных в интерфейсе

```
interface math_bus
#(parameter type DTYPE = int)
(input logic clock);
DTYPE a, b, result;//Параметризованный тип
task Read (output DTYPE a, b);
... // Считывание значений a и b
endtask
modport intjo (import Read,
input clock, output result);

modport fpjo (import Read,
input clock, output result);
endinterface

module top (input logic clock, resetN);

math_bus bus_a(clock); // Использование данных int
math_bus (#,DTYPE(real)> bus_b(clock); // Использование данных real
integer_math_unit i1 (bus_a. intjo);

// Подключение к интерфейсу, использующему целочисленный тип
floating_point_unit i1 (bus b.intjo);
// Подключение к интерфейсу, использующему вещественный тип
floating_point_unit i2 (bus_b.fpjo);
endmodule // end of module top
```

Преимущества: применения конструкции interface

1) Интерфейс идеален для повторного использования. Когда для коммуникации применяется протокол, сигналы повторяются.
2) Интерфейс собирает множество разнообразных сигналов, которые декларируются в различных модулях или программах, и размещает их в одном
месте, что уменьшает возможность ошибки неправильного подключения сигналов.

3) Чтобы добавить новые сигналы, достаточно объявить их только в интерфейсе, что также позволяет минимизировать вероятность ошибки.
4) Modport позволяет модулю более простым способом связывать сигналы с интерфейсом. Можно описывать направление сигнала для дополнительной проверки.

Недостатки:
1) Для соединения point-to-point интерфейс, использующий modport, имеет почти такой же большой размер, как и применение портов в списке сигналов. Однако все декларации находятся в одном месте, уменьшая вероятность внесения ошибки.
2) Необходимость использования имени интерфейса в дополнение к имени сигнала, что увеличивает размер текста модуля.
3) Если два блока проекта соединяются с помощью одного протокола, который не будет повторно использоваться, то построение интерфейса потребует больше усилий, чем непосредственное соединение портов.
4) Достаточно сложно выполнить соединение двух различных интерфейсов.

БЛОК «PROGRAM»

ПРИМЕЧАНИЕ-модуль является основным строительным блоком в Verilog. Модули могут содержать иерархии других модулей, связи , задачи и функции. декларации и присваивания в рамках процедурных ALWAYS и INITIAL блоков. Эта конструкция очень хороша для описания аппаратного обеспечения. Однако, для testbench, акцент

делается не на аппаратном уровне - таких деталей , как провода, структурная иерархия и соединения, а в моделировании полной среды, в которой устройство работает . Много усилий тратится на то чтобы оборудование было правильно инициализировано и синхронизированы, чтобы исключить гонки
между DUT и testbench, автоматизировать генерирование входных стимулов, обеспечить повторное использование существующих моделей и других инфраструктур.
Программный блок служит трем основным целям:
- Он предоставляет точку входа для выполнения тестов.
- Создается объем, который инкапсулирует program wide данные, задачи и функции.
- Он обеспечивает синтаксический контекст, который определяет планирование .
Программа служит ясным разделителем между DUT и testbench, и, что более важно, она определяет специализированные выполнения семантики для всех элементов, объявленных в программе. Вместе с синхронизирующим блоком, программа позволяет обеспечить без гоночное взаимодействия между DUT и testbench и
позволяет создавать циклы и на уровне абстрактных транзакций.
Абстракции и модельные конструкций SystemVerilog упрощают создание и поддержание тестов.

16,2 Построение программы

Типичная программа содержит объявления типов и о данных, подпрограммы, подключение к DUT, а также один или более потоков процедурных команд. Связи между DUT и testbench использует те же механизмы соединения, используемые SystemVerilog : указывается порт соединения, в том числе интерфейсы. Синтаксис
блока Программа выглядит следующим образом:

program_ansi_header ::=
{attribute_instance } program [lifetime] program_identifier
[parameter_port_list] [list_of_port_declarations] ;
program_declaration ::=
program_ansi_header [timeunits_declaration] { non_port_program_item }
endprogram [: program_identifier]
| { attribute_instance } program program_identifier (.*) ;
[timeunits_declaration] { program_item }
endprogram [: program_identifier]
| extern program_nonansi_header
| extern program_ansi_header
program_item ::=
port_declaration ;
| non_port_program_item
non_port_program_item ::=
{ attribute_instance } continuous_assign
| { attribute_instance } module_or_generate_item_declaration
| { attribute_instance } initial_construct
| { attribute_instance } final_construct
| { attribute_instance } concurrent_assertion_item
| { attribute_instance } timeunits_declaration17
| program_generate_item
program_generate_item37 ::=
loop_generate_construct
| conditional_generate_construct
| generate_region
lifetime ::= **static** | **automatic**
anonymous_program ::= **program ;** { anonymous_program_item } **endprogram**
anonymous_program_item ::=

task_declaration
| function_declaration
| class_declaration
| covergroup_declaration
| class_constructor_declaration
| **;**

Примеры

```
program test (input clk, input [16:1] addr, inout [7:0] data);
initial ...
endprogram
```

or

```
program test ( interface device_ifc );
initial ...
endprogram
```

Хотя программы это новый элемент для SystemVerilog, его включение является естественным продолжением.

Программу можно трактовать как модуль со специальной семантикой выполнения. После объявления программный блок может быть INSTANTIATED в нужном месте иерархической структуры (как правило, на верхнем уровне), а его порты могут быть подключены таким же образом, как и любой другой модуль.

блоки «Программа» могут быть вложенными внутри модулей и интерфейсов. Это позволяет нескольким сотрудничающим программам

разделять локальные для конкретной области переменные .

```
interface bus ();
//состав цепей интерфейса
  logic check_point;
  logic [2:0] data_in;
     logic [2:0] data_out;
//режимы интерфейса для тестирующих
// блоков
modport stim
(output data_in, data_out, check_point);
modport monit
(input data_in, data_out, check_point);
//так может выглядеть режим интерфейса
//для испытуемого устройства
 modport syst
(input data_in, output data_out);

endinterface
```

```
 module mot (  bus.monit uuu);
  parameter truth_table =8 'b 10001001;
    logic vt;
    logic error;
   assign  vt =truth_table[uuu.data_in] ;
   always @ (posedge uuu.check_point)
      error = uuu.data_out[0]== vt ?  0:1;
endmodule

program stimul (bus.stim uuu);
      parameter delay = 6;
      integer j;
      initial begin
        uuu.check_point=0;
         for (j=0; j<=7;j=j+1) begin
        #10 ;
           uuu.data_in=j;
         uuu.check_point= # delay 1;
            #3 ;
           uuu.check_point=0;
                                end
                    end
 endprogram
```

```
module test_sv;
    logic [2:0] data_in;
    logic [2:0] data_out;
    logic chech_point;
    bus  uuu (); //   оператор вхождения блока типа  bus, uuu- имя входения
                  //  список соответствий здесь не нужен потому, что имена фактических и
                  //формальных параметров совпадают
   mot  m0 (.uuu(uuu));
  stimul #6  p2(.uuu);
   complex
    m1(uuu.data_in,uuu. data_out[0],uuu.data_out[1],uuu.data_out[2]);
 endmodule
```

Реализация межпрограммного интерфейса

Обычно после разделения функций между аппаратной и программной частью проекта для обоих частей разрабатываются детальные программы в соответствующих языках. Сегодня, несмотря на наличие универсальных языков описания программных и аппаратных реализаций (SystemC, HandlerC), в большинстве проектных групп программы создаются и компилируются независимо. Программа, назначенная в процессорный модуль, пишется на одном из языков программирования, и после компиляции загружается в память процессорного блока. Фрагменты, назначенные для аппаратной части, пишутся на языке проектирования аппаратуры (ЯПА) и после компиляции при использовании программируемой логики загружается в конфигурационную память микросхемы, а если создается заказная микросхема, то описание

интерпретируется средствами подготовки соответствующего технологического процесса. В реальной аппаратуре взаимодействии обеих частей обеспечивается создаваемым физическим интерфейсом. Подобно, для комплексной проверки поведения целостной системы до ее физического исполнения необходимо использовать модель взаимодействия в форме межпрограммного интерфейса.

Но даже при создании «чисто аппаратных» проектов совместное использование традиционных языков программирования и языков проектирования аппаратуры может быть целесообразно, особенно для создания программ тестирования. Это полезно как с точки зрения использования больших и более гибких в сравнении с большинством ЯПА логических возможностей языков программирования, так и с точки зрения привлечения специалистов, способных взглянуть на проект с иной по сравнения с проектировщиком аппаратуры точки зрения, абстрагироваться от схемных и структурных деталей проекта и сосредоточиться на проверке именно алгоритмического соответствия проекта его спецификации.

Для обеспечения связи между программами, написанными на разных языках необходимо устанавливать средства их взаимодействия, иначе «межпрограммный интерфейс». Разработан целый ряд средств обеспечения такого взаимодействия: Programming lanquage interface (PLI), Verilog Programming interface (VPI), VHDL Procedural Interface (VHDL PLI) [12, 24, 37, 56, 61]

Стандарт IEEE1800 определяет общий принцип взаимодействия программ на языке SystemVerilog с программами, написанными на других языках, названный DPI (Direct Programming Interface, в переводе «Прямой Программный Интерфейс»). В данном разделе остановимся на правилах использования этого инструмента для связи SV программ с программами на языке C.

DPI обеспечивает простой, простой и эффективный способ подключения SV- и C-кодов, и позволяет создавать устройства и тестовые программ с использованием компонентов, описанных на SystemVerilog и C. Текущая версия DPI допускает его использование также для объединения фрагментов на языках SV и SystemC.

DPI определяет два уровня описания (Layer) – уровень SystemVerilog и уровень присоединяемого языка (в стандарте foreign). Эти уровни изолированы. Компилятор присоединяемого языка не используется для компиляции SV, и наоборот SV-компилятор не оперирует напрямую с программами на других языках. Разделение и взаимодействие базируется на использовании подпрограмм как замкнутых программных единиц. В общем случае подпрограмма рассматривается как черный ящик и может реализоваться представляться как в SV-уровне, так и на уровне присоединяемого языка, причем способ вызова не зависит от реализации. При использовании DPI SystemVerilog задача или функция может напрямую вызывать C или SystemC функции. Подпрограммы, передаваемые через DPI, трактуются как исполняемые мгновенно. Не поддерживается иных способов синхронизации процессов, кроме инициализации через изменение

аргументов. Функции, реализованные в C или SystemC называются импортируемыми функциями. Импортируемая функция в SV-программе перед вызовом должна быть продекларирована выражением **import**.

Декларация **import** задает имя подпрограммы (задачи или функции) тип возвращаемых данных (для функции), а также направления передачи (input, output или inout) формальных аргументов. Число аргументов должно совпадать с количеством аргументов в C или SystemC подпрограмме, а типы данных для аргументов должны быть совместимы с типами данных и функций C или SystemC. Функции могут иметь возвращаемое значение, или не возвращать (иметь тип **void)**.

Декларация i**mport** определяет задачи или функции в области, в которой она записана. Недопустимо определять импорт одной и той задачи или функции в одном модуле несколько раз. Но одна C или SystemC функция может быть импортирована в несколько модулей. Импортируемые задачи и функции вызываются тем же способом, что собственные SystemVerilog задачи и функций и неотличимы от вызовов задач или функций в SV. В следующем примере функции C именем hello() объявляется в SystemVerilog модуле декларацией i**mport**, а затем она вызывается.

Упрощенная форма декларации подпрограммы, импортируемой из Си в SV, имеет вид:

import [“DPI”|"DPI-C"][pure|context][cname=] <named_function_proto>;

Здесь <named_function_proto> – это прототип подпрограммы для SV слоя, cname – имя и тип импортируемой функции со стороны Си слоя. Если имя подпрограммы для SV-слоя совпадает с именем Си подпрограммы, это указание не обязательно.

Кроме имени подпрограммы указывается вид подпрограммы (task или function), список фактических аргументов, а для функции также тип возвращаемого значения.

Пример 1.9. Декларация и использование импортируемой программы

C layer	**SV layer**
	module top
	import “DPI” task hello();
#include stdio. h	. . .
void hello()	**initial**
{printf (“C-function started”);}	**if**(sig==1) hello();
	. . .
	endmodule

Декларации “DPI” и "DPI-C" для объявления связи с Си равноценны. Для связи с другими языками предполагается идентификация исходного языка, например "DPI-SC" для связи с SystemC. Но в настоящее время связь с другими языками помимо Си и SystemC не поддерживается.

Рассмотрим выражение

import “DPI” calc_parity_func **= function int** calc_parity **(input int** a**);**

Здесь определено, что модуль может вызывать функцию `calc_parity` анализа входного параметра “a”, представленного целым, на четность, которой на C - уровне соответствует функция `calc_parity_func`.

Следующая декларация объявляет, что импортируемая из C-уровня подпрограмма с именем `calc_task` в SV слое трактуется как задача с тем же именем.

import "DPI-C" task calc_task **(input int** in1**, output int** out1**);**

Подпрограмма SV уровня может исполняться на C-уровне. Такая подпрограмма называется экспортируемой. В SV программе определяется ее функционирование по обычным для SV правилам. Причем эта же подпрограмма может вызываться и на SV уровне. Для ее передачи SV программа должна содержать декларацию экспорта

export [“DPI”|"DPI-C"][context][cname=] <named_function_proto>;

Смысл разделов декларации такой же, как для декларации импорта.

На стороне C-уровня функция должна быть объявлена как внешняя (extern) по обычным для C правилам. В следующем примере модуль Bus содержит 2 функции: SV функцию `write`, которая может быть экспортирована на C уровень, и функцию `slave_write` которая импортируется из C.

Пример 1.10. Импорт и экспорт подпрограмм

SV- уровень

```
module Bus(input In1, output Out1);
  import "DPI" function void  slave_write
(input int address,   input int data);

 export "DPI" function write; /* Эта SV
функция может вызываться на С-слое*/
function void write(int address, int data);
// вызов С функции
 slave_write(address, data);
  endfunction
 ...
endmodule
```

Си - уровень

```
#include "svdpi.h"
extern void write(int, int);
// импортируется из  SystemVerilog

void slave_write(const int I1, const int I2)
{
 buff[I1] = I2;
...      }
```

Импортируемые Си и SystemC функции могут быть объявлены как «pure» (чистые) или «context» (контекстные). Импортируемая функция может быть определена как чистая, если результат ее исполнения зависит только от значений входных аргументов, перечисленных в интерфейсном списке прототипа. Чистые функции не могут иметь параметры типа out и inout и не могут быть void- функциями.

Декларация функции, как чистой, часто приводит к улучшению производительности моделирования, потому что допускает большую оптимизацию.

Если предполагается, что импортируемая подпрограмма (как task, так и function) может, в свою очередь, вызывать экспортируемые подпрограммы или использовать данные SV уровня, не определенные в интерфейсном списке (в том числе использовать вызовы через иные межпрограммные интерфейсы, такие как PLI), то такая подпрограмма должна быть определена как контекстная.

Пример 1.11 Декларации чистой и контекстной функций

```
import "DPI-C"  pure  function int calc_parity (input int a);
import "DPI-SC" context function int myclassfunc_func1 ( );
```

Контекстные задачи и функции, обрабатываются специальными средствами, что ограничивает оптимизационные возможности компилятора. Скорость моделирования уменьшаться. Не следует определять подпрограмму как контекстную, когда в этом нет необходимости.

Экспортируемая подпрограмма всегда имеет тип context.

DPI обеспечивает возможность использования для возвращаемых значений и аргументов большинство типов данных, определенных в SystemVeriliog:

- скалярные: `bit`, `logic`, `byte`, `shortint`, `int`, `longint`, `real`, `string` и `chandle`,

– одномерные и многомерные массивы данных типа bit и logic,

– неограниченные массивы типов bit и logic, а также bit-vector и integer,

и ряд других [56].

Соответствие базовых типов SV и C представлено в таблице 1.1.

Таблица 1.1

Соответствие типов данных языков C и SystemVerilog

SV типы данных	C-типы данных
byte	char
int	int
longint	long int
real	double
shortreal	float
chandle	void*
string	char*

Соответствие перечисленных в таблице типов определено для всех SV моделировщиков на любых платформах в файле svdpi.h. Эти преобразования руководства пользователя определяют как “implementation independent”. Если такие типы используются, следует присоединить файл svdpi.h к C – программе, используя декларацию #include.

Стандарты языков C и C++ не предполагают данных в четырехзначном алфавите, а также упакованных агрегатных типов данных: arrays (массивы), structures (стуктуры), unions (объединения). Для того чтобы система моделирования SV воспроизводила требуемое поведение данных таких типов при их обработке на Си-уровне, следует присоединять к C-программе файл svdpi_src.h. Приложения, требующие использования этого средства, не являются бинарно-совместимыми (являются платформно зависимыми на уровне исполнения), но совместимы на уровне исходного кода.

Пользователь имеет возможность реализовать связь также и с данными иных типов, в том числе им определяемых. Для этого необходимо доопределять правила преобразования, вводя соответствующие файлы – заголовки (header – файлы).

Пример 1.12 иллюстрирует реализацию связи Си и SV уровней при использовании различных типов данных.

Пример 1.12 Связывание данных C и SystemVerilog

SV- уровень

```
typedef struct {
 byte A;
 bit [4:1][0:7] B;
 int C;
} ABC;
/* импорт из Си; функция определена как
контекстная, так как использует вызов
экспортируемой функции*/
import "DPI" context function void
                C_Func(input ABC S);
// экспорт в Си
export "DPI" function SV_Func;
function void SV_Func(input int In,
             output logic[15:0] Out);
//< содержательное описание функции>
endfunction
```

Си - уровень

```
#include "svdpi.h"
#include "svdpi_src.h"
typedef struct {
 char A;
 SV_BIT_PACKED_ARRAY(4*8, B);
// платформно зависимый объект
 int C;
} ABC;
SV_LOGIC_PACKED_ARRAY(64, Arr);
// платформно зависимый объект
// импорт из SystemVerilog
extern void SV_Func(const int,
             SVLogicPackedArrRef);
 void C_Func(const ABC *S)
   { ...
 /* Первый аргумент передается по
значению, второй по ссылке*/
SV_Func(2,(SVLogicPackedArrRef)&Arr);
  }
```