. 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] Direct programming interface DPI

PROGRAMMING LANGUAGE INTERFACE

Используется для

генерации тестов, отладки фрагментов программы;

– построения различных программных приложений: трансляторы, анализ временных параметров;

– получения информации о проекте: иерархия, связи;

– специальных и пользовательских форм управления выводом информации;

– создания подпрограмм, формирующих тестовые последовательности;

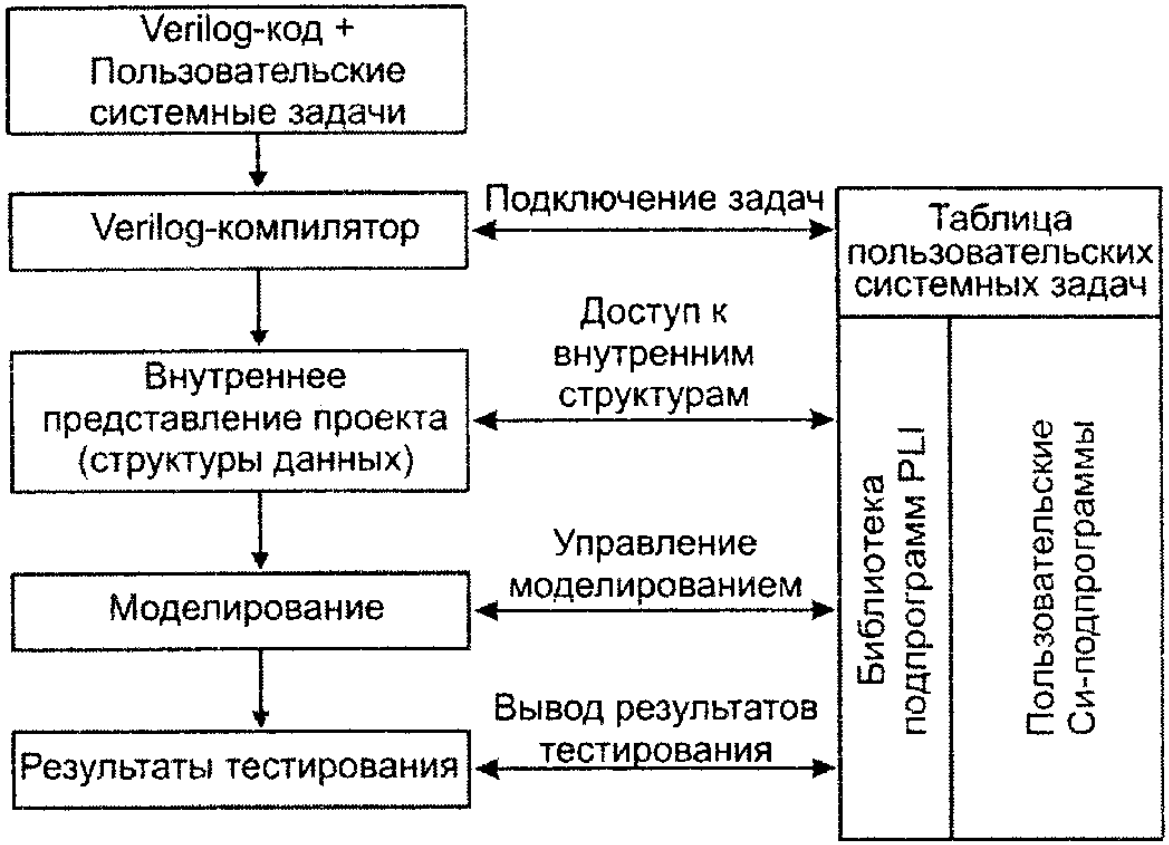

Рис. 8.1. PLI-интерфейс

PLI позволяет пользователю читать и модифицировать структуры внутренних данных, получать доступ к среде моделирования.

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 **import** 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 **import**, 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 С - уровне соответствует функция `calc_parity_func`.

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

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;
...      }
```

SV- уровень

```
//  TEST.SV

module test ();
typedef enum {RED, GREEN, YELLOW}
traffic_signal;

traffic_signal light;
function void sv_GreenLight ();
            begin
                  light = GREEN;
            end
endfunction

function void sv_YellowLight ();
          begin
                  light = YELLOW;
          end
endfunction

function void sv_RedLight ();
          begin
                  light = RED;
          end
endfunction

task sv_WaitForRed ();
        begin
          #10;
        end
endtask

export "DPI-C" function sv_YellowLight;
export "DPI-C" function sv_RedLight;
export "DPI-C" task sv_WaitForRed;

import "DPI-C" context task c_CarWaiting ();

initial
begin
          #10 sv_GreenLight;
          #10 c_CarWaiting;
          #10 sv_GreenLight;
end

endmodule
```

Си - уровень

```
//FOREIGN.c
#include "dpi_types.h"

int c_CarWaiting()
{
   printf("There's a car waiting on the other side. \n");
          printf("Initiate change sequence ...\n");
          sv_YellowLight();
          sv_WaitForRed();
          sv_RedLight();
          return 0;
}
```

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 двухзначных 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 С – программе, используя декларацию #include.

Стандарты языков С и С++ 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 С-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);
```

}

Program lanquage interface

Является более универсальным и гибким средством стыковки программ

Сначала создается подпрограмма на языке Си с использованием библиотеки PLI:

```
#include "veriuser.h"
void welcome_task()
{       io_printf("You are welcome!\n");
}
int function_call()
{       io_printf("This PLI function works !\n");
        return 0;
}
```

Для того чтобы программа моделирования знала о существовании подпрограммы welcome_task и могла вызвать ее, встретив конструкцию $welcome_task, необходимо выполнить подключение PLI-подпрограммы к Verilog-симулятору. Разные программы моделирования могут иметь различный механизм линковки PLI. Симуляторы Aldec Active-HDL, Cadence Verilog-XL™, MTI ModelSim® предлагают механизм регистрации системных задач и функций с использованием массива veriusertfs. Его синтаксис:

```
s_tfcell veriusertfs[ ] =
{       {usertask | userfunction | userrealfunction, data; checktf; sizetf; calltf;
        misctf; "$tfname"},
{0}
}
```

Каждая задачи или функция должна иметь запись в таблице veriuserfs

При подключении каждого приложения моделировщик ищет экспортируемую таблицу **veriuserfs**

Листинг 8.1. Полная версия кода С из файла PliApp.c

```
#include "veriuser.h"
#include "aldecpli.h"
#include <windows.h>
int welcome_task()
{       io_printf("You are welcomet!\n");
        return 0;
}
int function_call()
{       io_printf("This PLI function works !\n");
        return 0;
}
extern "C" __declspec(dllexport)  s_tfcell veriusertfs[ ] =
{       {usertask, 0, 0, 0, welcome_task, 0, "$welcome_task"},
        {userfunction, 0, 0, 0, function_call, 0, "$my_function"},
        {0}
};
BOOL WINAPI DllMain( HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved )
{ return TRUE;
}
```

Листинг 8.2. Пример использования системной функции и задачи

```
module start_PLI;
initial
begin
        $welcome_task;
        $my_function;
end
endmodule
```

Подпрограммы библиотеки PLI

PLI-библиотека предлагает стандартный интерфейс для представления внутренних данных в проекте. Существует два класса PLI-подпрограмм: доступа (access routine) и обслуживания данных (utility routine) (vpi – не рассматривается).

Подпрограммы access routine обеспечивают доступ к представлению внутренних структур данных. Они позволяют просматривать и извлекать необходимую информацию о проекте. Обслуживающие подпрограммы (utility routine) главным образом используются для перемещения данных через Verilog/Programming Language границу и предоставляют пользователю разнообразные вспомогательные функции.

Существует шесть типов подпрограмм доступа:

1. Handle routine. Возвращает абстрактный идентификатор объекта в проекте. Имя таких подпрограмм всегда начинается с acc_handle_.

2. Next routine. Возвращает идентификатор следующего объекта из множества заданного типа объектов проекта. Всегда начинается с acc_next_ и получает ссылки на объекты в качестве аргументов.

3. Value Change Link (VCL) routine. Позволяет пользовательской системной задаче добавлять и удалять объекты из списка наблюдаемых. Всегда начинается с acc_vcl_ и не возвращает значение.

4. Fetch routine. Извлекает разнообразную информацию об объекте: полное имя иерархического пути, относительное имя. Начинается с acc_fetch_.

5. Utility access routine. Выполняет вспомогательные операции для подпрограмм доступа. Например, acc_initialize() и acc_close().

6. Modify routines. Может изменять внутренние структуры данных.

Обслуживающие подпрограммы (utility routine)

Задачи, решаемые с применением обслуживающих подпрограмм:

– получение информации о вызванных системных задачах Verilog и о списке аргументов. Считывание и изменение значения аргумента для вызванной системной задачи. Наблюдение за изменениями значений аргументов;

– выполнение вспомогательных задач, таких как сохранение рабочей области, указателя на задачу;

– выполнение сложных вычислений;

– вывод сообщения;

– получение информации о времени моделирования и очереди событий. Остановка, завершение, сохранение и восстановление процесса моделирования.

Подпрограммы этого тина имеют префикс tf_