Глава 3

Организация проекта

3.1. Подпрограммы

Декларация подпрограммы

Определение.

subprogram_specification ::=
procedure designator [(formal_parameter_list)]
| [**pure** | **impure**] **function** designator [(formal_parameter_list)]
return type_mark

Подпрограммы имеют две формы — функции и процедуры. Вызов процедуры есть оператор, в то время как вызов функции возвращает значение в выражении. Указателем функции (операторным символом) может быть идентификатор или строковый литерал. Заметим, что строковый литерал не может быть именем процедуры. Подпрограммы могут быть декларированы в тексте пакета, интерфейсе объекта проекта, архитектурном теле, процессе, процедуре или функции. Подпрограммы могут вызывать другие подпрограммы. Примеры подпрограмм уже приводились и будут даны далее.

3.2. Функции

Общий вид оператора *декларации* функции

[pure | impure] function имя функции (параметр {, параметр})
return тип возвращаемого функцией значения **is**
раздел деклараций

```
begin
        тело функции
end [имя функции];
```

Общий вид оператора *вызова* функции

имя функции (фактический параметр {, фактический параметр});

Функция имеет только входные параметры. Следующий иллюстративный пример показывает (декларирует) функцию BOOL_TO_SL, преобразующую тип BOOLEAN в тип STD_ULOGIC. Типы данных STD_ULOGIC, STD_LOGIC будут объяснены в разд. 3.4.

Пример.

```
function BOOL_TO_SL(X : boolean)
          return std_ulogic is
begin
  if X then
    return '1';
  else
    return '0';
  end if;
end BOOL_TO_SL;
```

Следующий пример показывает функцию преобразования типа bit в тип boolean. Напомним, что данные типы не эквивалентны.

```
function bit_bool (inp_bit : in bit) return boolean is
begin
if (inp_bit = '1') then
return true;
else
return false;
end if;
end bit_bool;
```

Функция может содержать последовательные операторы, исключая операторы ожидания и назначения сигналов. В теле функции могут декларироваться локальные переменные. Например, в теле функции PARITY декларируется локальная переменная TMP.

Пример.

```
function PARITY (X : std_ulogic_vector)
            return std_ulogic is
 variable TMP : std_ulogic := '0';
begin
 for J in X'range loop
  TMP := TMP xor X(J);
 end loop;
 return TMP;
end PARITY;
```

Пример использования функции PARITY в архитектурном теле.

```
architecture FUNCTIONS of PAR is
begin          -- вызов функции
 PARITY_BYTE <= PARITY(DATA_BYTE);
 PARITY_WORD <= PARITY(DATA_WORD);
end FUNCTIONS;
```

Необязательные ключевые слова pure, impure при декларации функции имеют следующий смысл: ключевое слово **impure** говорит о том, что функция может иметь побочные эффекты, т. е. такая функция может иметь доступ к внешним данным, например, читать внешний файл [7]. Функции без побочных эффектов могут декларироваться с необязательным словом **pure**. В данной книге все функции языка VHDL являются функциями без побочных эффектов.

3.3. Процедуры

Общий вид оператора *декларации* процедуры

procedure имя процедуры (параметр {, параметр}) **is**
    раздел деклараций

begin
тело процедуры
end [имя процедуры];

Процедура может иметь входные (**in**), выходные (**out**) и вход/выходные (**inout**) параметры. Это могут быть сигналы, переменные или константы. По умолчанию входные параметры — константы, выходные и вход/выходные параметры — переменные.

Общий вид оператора *вызова* процедуры

имя процедуры (фактический параметр {, фактический параметр});

Таким образом, вызов процедуры состоит из имени процедуры и списка фактических параметров. Процедуры могут вызываться последовательно или параллельно. При параллельном вызове происходит выполнение процедуры, когда какой-нибудь входной вход/выходной параметр изменился. VHDL-код с текстом подпрограммы, т. е. функции или процедуры может находится в разделе деклараций архитектурного тела, либо в пакете. Если подпрограмма определена в пакете, то текст подпрограммы должен быть в теле пакета.

Если подпрограмма находится в пакете, то в этом случае перед текстом entity, где процедура используется (процедуры могут использоваться в архитектурных телах), требуется указание имени библиотеки и имени пакета, в котором содержится текст подпрограммы. Библиотеки и содержащиеся в них пакеты будут рассмотрены далее.

Следующий пример процедуры PARITY показывает отличия процедуры от функции: функция имеет только входные параметры, режим (mode) которых не специфицируется, в процедуре могут содержаться операторы назначения сигналов.

Пример.

```
procedure PARITY
  (signal X : in std_ulogic_vector;
   signal Y : out std_ulogic) is
  variable TMP : std_ulogic := '0';
```

```
begin
 for J in X'range loop
  TMP := TMP xor X(J);
 end loop;
 Y <= TMP;        -- в процедуре могут быть операторы
                  --  назначения сигналов
end PARITY;
```

Следующий пример показывает процедуру DISPLAY_MUX и ее вызов в архитектурном теле SUBPROG.

```
procedure DISPLAY_MUX
 (ALARM_TIME, CURRENT_TIME : in digit;
  SHOW_A           : in std_ulogic;
  signal DISPLAY_TIME : out digit) is
begin
 if (SHOW_A = '1') then
  DISPLAY_TIME <= ALARM_TIME;
 else
  DISPLAY_TIME <= CURRENT_TIME;
 end if;
end DISPLAY_MUX;
architecture SUBPROG of DISP_MUX is
 ...
begin        -- вызов процедуры
 DISPLAY_MUX (ALARM_TIME, CURRENT_TIME,
          SHOW_A, DISPLAY_TIME);
end SUBPROG;
```

3.4. Разрешающие функции. Пакет std_logic_1164

Когда сигнал имеет один драйвер (иногда драйвер называют контейнером), то значение сигнала легко определить, потому что, когда процесс приостанавливается, значение сигнала из драйвера передается сигналу. Однако во многих практических ситуациях один и тот же сигнал может назначаться в различных (нескольких) процессах. Например, шина данных в персональном компьютере может получать данные из процессора, памяти (ОЗУ), жесткого диска и других устройств. Сигнал от многих источников называется в

языке VHDL *разрешаемым* (resolved), для таких сигналов требуется написание функции разрешения (разрешающей функции). *Разрешающая функция* — это функция определения значения сигнала по его значениям из различных источников. Далее будем рассматривать два источника и для этого случая изучать функции разрешения в электронных схемах. Для моделирования процессов прохождения сигналов в реальных электронных схемах используется *многозначная логика*. Тип bit обобщается на случай девяти значений сигнала

'U' -- не инициализировано;
'X' -- неизвестное значение (сильный источник сигнала);
'0' -- логический 0 (сильный источник сигнала);
'1' -- логическая 1 (сильный источник сигнала);
'Z' -- высокий импеданс (цепь не подключена к источнику);
'W' -- неизвестное значение (слабый источник сигнала);
'L' -- логический 0 (слабый источник сигнала);
'H' -- логическая 1 (слабый источник сигнала);
'-' -- безразличное значение (don't care).

На практике чаще всего используется шестизначный алфавит {U, X, 0, 1, Z, –}. Заметим, что неизвестное значение 'X' не эквивалентно неопределенному значению '–'. Безразличное значение эффективно используется при логическом синтезе схемы и логической оптимизации. Читатель, знакомый, например, с методами минимизации не полностью определенных булевых функций, может вспомнить о том, что некоторые безразличные значения '–' функции заменяются при минимизации определенными (0, 1) с целью получения лучшего результата. Расширение типа bit на случай девяти значений сигнала привело к понятию типа ***std_logic***. Основное назначение типа std_logic — это повысить точность моделирования и дать «легальную» возможность разработчику делать многократные присваивания одному и тому же сигналу.

Пример.

```
Library IEEE;
use IEEE. std_logic_1164. all;
entity test_flag is
end;
```

```
architecture beh of test_flag is
Signal FlagC : std_logic := 'Z';   signal Carry : boolean;
begin
ALU: process (carry)
begin
if  Carry then FlagC <= '1'; end if;   end process ALU;
COMM: process (carry)
        begin
            FlagC <= 'Z';
end process COMM;
Carry <= true, false after 100 ns, true after 200 ns;
end beh;
```

В данном примере для сигнала FlagC не требуется разрешающая функция, хотя операторы назначения данного сигнала имеются в двух процессах (рис. 3.1).

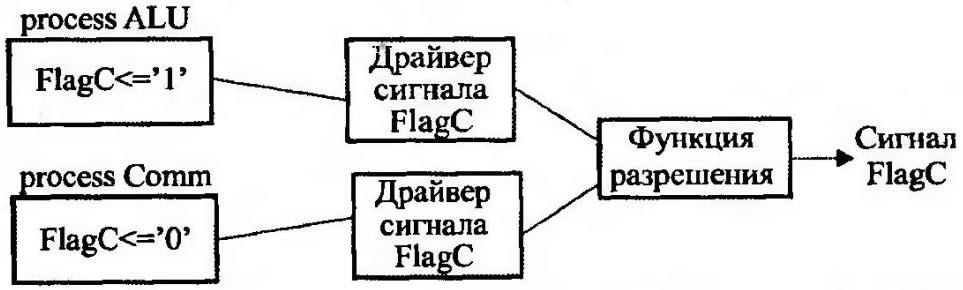

Рис. 3.1. Функция разрешения для сигналов с многими драйверами

Легко проверить моделированием, что сигнал FlagC будет иметь значение '1'. Почему же в данном примере не требуется написание разрешающей функции? Дело в том, что для сигналов типа std_logic допустимы присваивания из нескольких источников, и при этом не требуется написание разрешающей функции — данная функция находится в пакете STD_LOGIC_1164, который всегда требуется указывать, если используется тип std_logic и, естественно, тип std_logic_vector.

! *Типы std_logic, std_logic_vector являются* ***разрешаемыми*** *типами, для сигналов данного типа допустимы присваивания из нескольких источников.*

Тип std_ulogic (и тип std_ulogic_vector) является *не разрешаемым* перечислимым типом с множеством значений {'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '–'}. Для сигналов типа std_ulogic **запрещено** использовать много источников сигнала, поэтому требуется написание собственной разрешающей функции. Тип std_logic называется разрешаемым типом, он является подтипом типа std_ulogic. Буква u в названии типа std_ulogic сигнализирует о термине unresolved (не разрешаемый).

Итак, типы std_logic, std_ulogic — это перечислимые типы. Их определение содержится в пакете **std_logic_1164** (см. приложение 4).

В пакете std_logic_1164 содержатся определения типов, подтипов, функций на случай многозначной логики:

- тип std_ulogic — не разрешаемый логический тип с девятизначным алфавитом;
- тип std_logic_vector;
- подтип std_logic типа std_ulogic;
- «перегружаемые» логические функции and, nand, or, xor, not для операндов std_ulogic и др.

Заметим, что для преобразования типа bit в тип std_logic (и наоборот) требуются соответствующие функции. Табл. 3.1, по существу, есть табличное задание разрешающей функции F для сигнала из двух источников. Рис. 3.2 иллюстрирует смысл табл. 3.1, т. е. получение значения сигнала с помощью функции разрешения F.

Таблица 3.1

	U	X	0	1	Z	W	L	H	-
U	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
X	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
0	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
а 1	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
Z	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
б W	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
L	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
в H	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
-	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

Рис. 3.2. Иллюстрация табличного задания функции разрешения F для сигналов типа std_logic с двумя драйверами

Фактически типы std_logic, std_ulogic стали промышленным стандартом и доступны во всех системах моделирования с языком VHDL, поставляющихся вместе с пакетом std_logic_1164. Чтобы использовать пакет std_logic_1164, необходимо указать следующее:

Library IEEE;
use IEEE. std_logic_1164. all;

Опишем схему (рис.3.3), используя для сигналов тип std_logic.

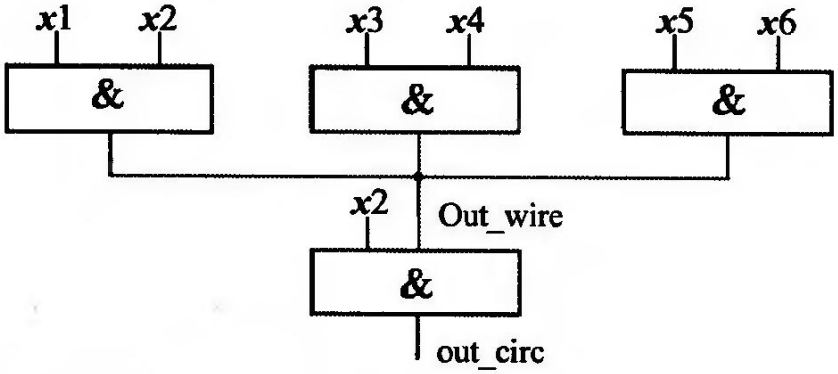

Рис.3.3. Схема circuit_wire

В VHDL-коде используются следующие имена: схема, изображенная на рис. 3.3, имеет имя circuit_wire, двухвходовый элемент И — имя cc.

library IEEE;
use IEEE.std_logic_1164.**all**;
library work;

```
use work.wire.all;
entity circuit_wire is
port (x1, x2, x3, x4, x5, x6 : in std_logic;
out_circ : out std_logic);
end circuit_wire;
architecture structure of circuit_wire is
component cc
port (x1, x2 : in std_logic; y : out std_logic);
end component;
signal out_wire : std_logic ;
begin
p1: cc port map (x1 => x1, x2 => x2, y => out_wire);
p2: cc port map (x1 => x3, x2 => x4, y => out_wire );
p3: cc port map (x1 => x5, x2 => x6, y => out_wire );
p4: cc port map (x1 => out_wire, x2 => x2, y => out_circ);
end structure;
library IEEE;
use IEEE.std_logic_1164.all;
entity cc is                    -- описание логического элемента И
port (x1, x2: in std_logic;  y: out std_logic);   end cc;
architecture functional of cc is
begin
y <= x1 and x2;
end functional;
```

Если же в описании сигналов схемы (рис. 3.3) использовать тип bit, то для описания монтажного ИЛИ требуется разрешающая функция. Введем тип resolved_bit (подтип типа bit) и определим в пакете wire, находящемся в рабочей библиотеке work, разрешающую функцию res_func для сигнала типа resolved_bit. Разрешающая функция определяет значение сигнала по значениям сигналов из нескольких источников. В данном примере разрешающая функция назначит сигналу out_wire значение '1', если хотя бы один из выходных сигналов логических элементов И первого уровня схемы (рис. 3.3) будет иметь значение '1', в противном случае сигнал out_wire примет значение '0'.

Ниже приведен пакет wire и тело (body) этого пакета.

```
package wire is
function RES_FUNC(DATA: in bit_vector) return bit;
subtype RESOLVED_BIT is RES_FUNC bit;  end;
package body wire is
 function RES_FUNC(DATA: in bit_vector) return bit is
begin
    for I in DATA'range  loop
        if DATA(I) = '1' then
 return '1';
        end if;
end loop;
 return '0';
end;
end;
```

В тексте функции RES_FUNC используется атрибут DATA'range — диапазон сигнала DATA. Описание схемы (с собственной разрешающей функцией) будет иметь вид.

```
library WORK;
use WORK.wire.all;
entity circuit_wire is
port (x1, x2, x3, x4, x5, x6 : in bit; out_circ : out bit);
end circuit_wire;
architecture structure of circuit_wire is
component cc
port (x1, x2 : in bit; y : out bit); end component;
signal out_wire : RESOLVED_BIT;
begin
p1: cc  port map (x1 => x1, x2 => x2, y => out_wire);
p2: cc port map (x1 => x3, x2 => x4, y => out_wire );
p3: cc port map (x1 => x5, x2 => x6, y => out_wire );
p4: cc port map (x1 => out_wire, x2 => x2, y => out_circ);  end;
```

3.5. Архитектура

Ранее в общем виде была представлена архитектура объекта проекта и интерфейс объекта проекта. Приведем определение архитектуры.

Определение.

architecture_body ::=
architecture identifier **of** *entity*_name **is**
architecture_declarative_part
begin
architecture_statement_part
end [architecture] [*architecture*_simple_name] ;

В разделе деклараций архитектуры (architecture_declarative_part) могут быть

- декларации подпрограмм;
- тела подпрограмм;
- спецификации конфигурации;
- декларации типов, подтипов, констант, сигналов, файлов, альтернативных точек входа, компонентов и др.

! В разделе деклараций архитектуры не могут быть декларированы локальные переменные.

! Только параллельные операторы размещаются в теле архитектуры между ключевыми словами **begin, end.** Последовательные операторы могут быть внутри оператора процесса или подпрограммы.

Допустимо, чтобы параллельных операторов не было внутри архитектурного тела.

3.6. Декларация интерфейса объекта

Определение.

entity_declaration ::=
entity identifier **is**
entity_header
entity_declarative_part
[**begin**
entity_statement_part]
end [**entity**] [*entity*_simple_name] ;

Необязательный раздел операторов **entity**, начинающийся с ключевого слова begin, может иметь только параллельные вызовы процедур, параллельные операторы сообщения, операторы процесса. Все они должны быть пассивными, так чтобы не было присваивания значений сигналам.

Раздел entity_header включает в себя список настройки и список портов:

[**generic** (generic_list);]
[**port** (port_list);]

Список портов (**port_list**) и список настройки (**generic_list**) такие же, как и в подпрограммах.

Если специфицировать объект другого класса, а не сигнал после ключевого слова **port**, то это является ошибкой. В **entity** ключевое слово **port** является необязательным.

Список после ключевого слова **generic** (общий, настраиваемый) есть хороший способ проведения (передачи) параметров в интерфейс объекта проекта, таких как параметры временной задержки, ширина шины (число проводников) и т. д.

Употребляя настройку (слово **generic**), можно специфицировать регулярную структуру переменной длины.

Приведем пример [9] **entity** для сдвигового регистра переменной длины N, определив сперва модификацию D-триггера с предустановкой (очисткой).

```
library IEEE;
use IEEE.std_logic_1164.all;
entity DFF is                          -- D-триггер
  generic (
    PRESET_CLRn  : integer);
  port (
    RSTn, CLK, D : in  std_logic;
    Q            : out std_logic);
end DFF;
architecture RTL of DFF is
begin
  process (RSTn, CLK)
```

```
  begin
    if (RSTn = '0') then
      if (PRESET_CLRn = 0) then
        Q <= '0';
      else
        Q <= '1';
      end if;
    elsif (CLK'event and CLK = '1') then
      Q <= D;
    end if;
  end process;
end RTL;
```

Обращаем внимание на то, что в entity SHIFTN сдвигового регистра переменной длины N имеется декларируемый сигнал T и оператор сообщения **assert**.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity SHIFTN is          -- сдвиговый регистр длины N
  generic (
    PRESET_CLRn  : integer;
    N            : integer);
  port (
    RSTn, CLK, SI : in  std_logic;
    SO            : out std_logic);
  signal T : std_logic_vector(N downto 0);
  begin
    assert (N > 2) and (N < 33) report
      "N outside of range 3 to 32";
end SHIFTN;

architecture RTL1 of SHIFTN is
  component DFF
  generic (
    PRESET_CLRn  : integer);
```

```
    port (
      RSTn, CLK, D : in  std_logic;
      Q            : out std_logic);
    end component;
  begin
    T(N) <= SI;
    SO   <= T(0);
    g0 : for i in N – 1 downto 0 generate
      allbit : DFF
      generic map (PRESET_CLRn => PRESET_CLRn)
    port map (RSTn => RSTn, CLK => CLK, D => T(i + 1),
Q => T(i));
    end generate;
  end RTL1;
```

Обращаем внимание на то, что спецификация компонента DFF в архитектурном теле RTL1 совпадает с его определением в entity DFF.

3.7. Карта портов и карта настройки

Соединение VHDL-описаний осуществляется с помощью карт (**map**).

Для иерархического проекта порты низкоуровневых компонент (уровня i – 1) могут быть отображены в порты высокоуровневого объекта (уровня i) сигналами с учетом следующих ограничений.

Сигнал порта, имеющий режим (mode) in, может быть соединен с портом, имеющим режим in, inout, buffer (рис. 3.4).

На рис. 3.4 изображены только некоторые возможные соединения портов компонента с портами той подсхемы, в которую компонент входит.

Сигнал порта режима out может быть соединен с портом, имеющим режим out или inout, buffer. Сигнал порта режима inout, buffer может быть отображен (соединен) с портом, имеющим режим вида inout, buffer соответственно.

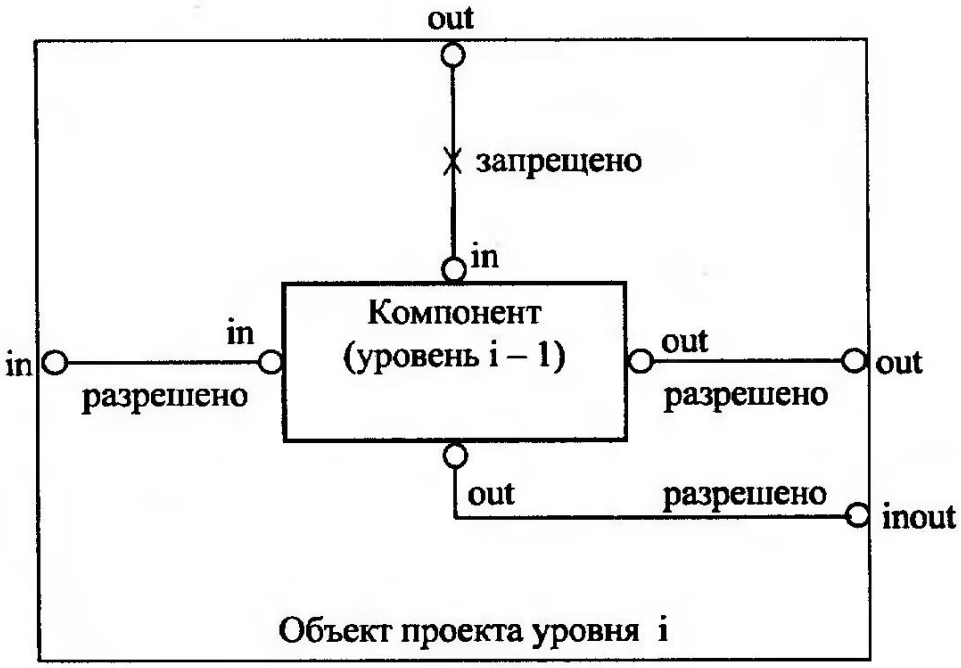

Рис. 3.4. Связи портов компонента с портами объекта

Сигнал порта режима linkage может быть соединен с сигналом порта любого режима. С точки зрения проектирования аппаратуры вид связи linkage не является естественным, данный тип связи более подходит для алгоритмических описаний.

Режим inout употребляется чаще всего для двунаправленных контактов, которые находятся снаружи кристалла СБИС. Для внутренних сигналов режим не указывается.

3.8. Конфигурация

Один и тот же объект проекта (entity) может иметь много архитектурных тел. Как при моделировании выбрать то или другое архитектурное тело?

Формальное определение (синтаксис) понятия конфигурации довольно сложное.

Начнем с простейших случаев

Пусть мы определили компоненту **goose**.

```
entity goose is
port (a, b: in bit; c : out bit);
end goose;
architecture struct_1 of goose is
        begin
c <= a xor b;
end struct_1;
```

Мы можем употребить данную компоненту стандартным образом, употребляя имя **goose**.

```
entity flock is
port (a, b: bit; c: out bit);
end flock;
architecture three_geese of flock is
signal w, r: bit;
component goose          -- декларация компонента
port (a, b: bit; c: out bit);
end component;
begin
one: goose port map (a, b, w);
                         -- создание экземпляра компонента
two: goose port map (a, w, r);
three: goose port map (a, r, c);
end three_geese;
```

Архитектурное тело содержит три экземпляра компонента goose. *По умолчанию* предполагается, что это тот же самый компонент goose, который был определен ранее и который содержится в рабочей библиотеке (**Work** — имя рабочей библиотеки).

Пусть у нас имеется другое архитектурное тело для компонента goose.

```
architecture struct_2 of goose is
        begin
c <= ((a and (not b)) or ((not a) and b)) ;
end struct_2;
```

Мы можем не действовать «по умолчанию», а «явно» определить употребление того или другого архитектурного тела для конкретного компонента на основе конфигурации.

```
configuration bird of flock is   -- bird — имя конфигурации
for three_geese
        for one: goose
                use entity work.goose(struct_2);
        end for;
for two: goose
                use entity work.goose(struct_1);
        end for;
for three: goose
                use entity work.goose(struct_1);
        end for;
end for;
end bird;
```

Мы указали, что для компонента goose, имеющего имя (метку) one, необходимо использовать (**use**) из рабочей библиотеки архитектурное тело struct_2, аналогично — для компонентов с метками two, three необходимо использовать архитектурное тело struct_1.

Рассмотрим немного более сложный пример конфигурации на примере схемы vlsi_1 (см. рис. 1.2). Представим, что в схемах сумматора adder_2 и умножителя mult_2 мы хотим использовать различные архитектурные тела для одноразрядного полусумматора add1. Итак, в умножителе будем использовать для подсхемы add1 архитектурное тело struct_2, а в сумматоре — архитектурное тело struct_1. Архитектурное тело struct_2 отличается от архитектурного тела struct_1 только назначением сигнала: вместо

```
s1 <= ((b1 and (not b2)) or (not b1) and b2));
```

можно использовать

```
s1<= (b1 xor b2);
```

Конфигурация example выглядит следующим образом:

```
configuration example of vlsi_1 is
 for structure  -- имя архитектурного тела схемы vlsi_1
  for circ2: mult_2
   use entity work.mult_2(structure);
   for structure
         for all: add1
        use entity work.add1(struct_2);
     end for;
   end for;
  end for;

  for circ3: adder_2
   use entity work.adder_2(structure);
   for structure
         for all: add1
        use entity work.add1(struct_1);
         end for;
         for all: add2
        use entity work.add2(struct_1);-- вместо struct_1
         end for;                      -- можно указать
end for;                               -- другую архитектуру
end for;
end for;
end example;
```

В данной конфигурации предполагается, что подсхемы сумматора и умножителя находятся в рабочей библиотеке work. Использование библиотек в языке VHDL рассматривается в следующем разделе.

При записи конфигурации могут быть использованы другие конфигурации для подсхем, входящих в схему, таким образом, описание конфигурации может быть организовано иерархически, наподобие иерархической организации описания проектируемой цифровой системы. Большое число примеров конфигураций для сдвигового регистра можно найти в [9].

3.9. Модули проекта и VHDL-библиотеки

Кроме объектов проекта (entity и архитектурных тел) VHDL-файл проекта может содержать и другие *модули проекта*. Используются следующая классификация: имеются *первичные* и *вторичные* модули проекта.

Первичные модули проекта:

- декларация объекта в целом (entity);
- декларация конфигурации;
- декларация пакета.

Первичный модуль может быть связан с многими *вторичными* модулями проекта, которыми могут быть:

- архитектурное тело;
- тело пакета.

! Каждый первичный модуль в данной библиотеке должен иметь уникальное простое имя. Каждое архитектурное тело, связанное с entity, должно быть уникальным.

Модули проекта (первичные и вторичные) размещаются в библиотеках (рис. 3.5). Рабочая библиотека (по умолчанию имеет имя WORK) может быть только одна.

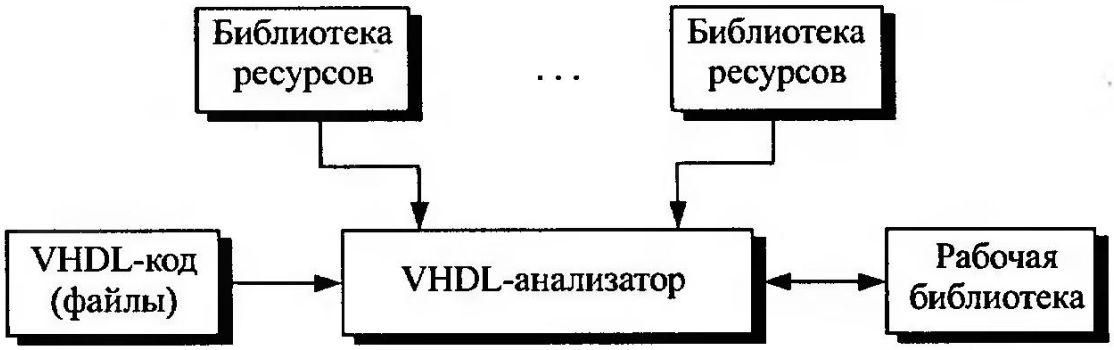

Рис. 3.5. VHDL-библиотеки и VHDL-анализатор

В *рабочей библиотеке* размещаются файлы (модули), анализируемые VHDL-анализатором. В *библиотеках ресурсов* размещаются модули, на которые ссылаются анализируемые блоки.

Ресурсной библиотеки может не быть во время анализа. Ссылка на библиотеку осуществляется указанием ключевого слова **library**. На-

пример, ссылка на библиотеку STD, содержащую пакет STANDARD, и на библиотеку WORK, содержащую пакет FXP, осуществляется следующим образом:

```
library STD, WORK;
use STD.STANDARD.all
use WORK.FXP.all
```

VHDL-анализатор читает файлы исходного VHDL-кода, читает ссылки на ресурсные библиотеки и генерирует базу данных моделирования в рабочей библиотеке.

Область видимости библиотеки — от начальной ссылки до конца декларативной области, связанной с блоком проекта, где ссылка появилась.

Видимость сигналов: сигнал, декларированный в пакете, является видимым во всех объектах проекта, которые употребляют ссылку (use) на данный пакет. Сигнал, декларированный в entity как порт, является видимым во всех архитектурных телах, связанных с данным entity. Сигнал, декларированный в разделе деклараций архитектурного тела, видим только внутри данного архитектурного тела.

Приведем пример VHDL-кода, в котором используется вызов функций, реализующих операторы сдвига. В данном случае функция sl реализует оператор sll (сдвиг влево), функция sr реализует оператор srl (сдвиг вправо). Функции sl, sr находятся в пакете exemplar_1164 библиотеки exemplar. Так как имеется ссылка на пакет exemplar_1164, то функции sl, sr являются *видимыми* (доступными), поэтому нет необходимости давать коды этих функций в разделе деклараций архитектурного тела. Так как используется тип std_logic_vector, то требуется также ссылка на пакет STD_LOGIC_1164.

```
library IEEE;
use IEEE.STD_LOGIC_1164. all;
library exemplar;
use exemplar.exemplar_1164.all;
entity shift_example is
```

```
port (a, b : in std_logic_vector (3 downto 0);
    x1, x2 : out  std_logic_vector (3 downto 0));
end shift_example;
architecture Synt of shift_example is
signal y, z : std_logic_vector (3 downto 0);
begin
y  <= sl (a, 3);              -- сдвиг влево
z  <= sr (a, 3);              -- сдвиг вправо
x1 <= y and b;
x2 <= z xor b;
end Synt;
```

В записи **sl (a, 3)** : **sl** — обозначение оператора (функции), **a** — массив бит, **3** — число разрядов, на которые надо сдвинуть влево массив **a**. В записи **sr (a, 3)** : **sr** — обозначение оператора (функции), **a** — массив бит, **3** — число разрядов, на которые надо сдвинуть вправо массив **a**. Данный пример показывает также, что логические операции можно выполнять поразрядно над массивами бит.

УПРАЖНЕНИЯ

1. В какой части VHDL-кода должны быть декларированы локальные сигналы архитектурного тела? Выберите правильный ответ:

a) в списке портов архитектурного тела;
b) в конфигурации;
c) в архитектурном теле после ключевого слова begin;
d) в архитектурном теле перед ключевым словом begin.

2. Как должны быть подобны компонент (component) и соответствующий интерфейс (entity)? Выберите правильный ответ:

a) entity и component должны иметь одно и то же имя, но порты могут различаться;

b) имена entity и component могут различаться, но имена портов должны быть одинаковы;

c) entity и component должны иметь одинаковые имена и должны иметь одинаковые имена портов.

3. Правильно ли, что компоненты, декларируемые в архитектурном теле, должны специфицироваться полностью, т. е. вместе с их интерфейсом и выполняемыми функциями?

4. Как много архитектурных тел может быть связано с одним entity? Выберите правильный ответ:

a) одно или более;
b) более одного;
c) только одно;
d) ни одного.

5. Правильно ли утверждение: «Каждый порт должен быть специфицирован с его режимом (mode)»?

6. Правильно ли утверждение: «Режим порта специфицирует направление потока данных через порт»?

7. Являются ли порты сигналами?

8. Правильно ли утверждение: «Описание каждого порта с комментарием в конце строки является необходимым согласно стандарту языка VHDL»?

9. Разрешается ли специфицировать начальное значение порта?

10. Может ли настраиваемый параметр (generic) динамически изменяться во время моделирования?

11. Правильно ли утверждение: «Все процессы в архитектурном теле являются активными, когда архитектура активна »?

12. Могут ли употребляться переменные для передачи информации между процессами?

13. Напишите VHDL-код для схемы 4-разрядного сумматора, представляющей каскадное соединение одноразрядных сумматоров. Напишите VHDL-код для одноразрядного сумматора add2 в виде

а) логической схемы элементов И, ИЛИ, НЕ;
б) логических функций.

Запишите различные конфигурации, используя различные архитектурные тела для одноразрядных сумматоров add1, add2.

14. Рассмотрите предыдущую задачу для случая 4-разрядного умножителя (см. рис. 2.6). Запишите различные конфигурации.

15. Разработайте логическую схему для вычисления функции

$$\underline{f} = (\underline{p} + \underline{q}) + \underline{w} \times \underline{d},$$

где $\underline{p} = (p_2, p_1)$, $\underline{q} = (q_2, q_1)$, $\underline{w} = (w_2, w_1)$, $\underline{d} = (d_2, d_1)$ — двухразрядные числа.

Опишите ее на языке VHDL. Запишите различные конфигурации, употребляя различные архитектурные тела для сумматоров и умножителя.

16. Что из перечисленного ниже не является модулем проекта? Выберите правильный ответ:

a) architecture;
b) entity;
c) process;
d) package.