Глава 2

Последовательные и параллельные операторы

2.1. Последовательные операторы

В VHDL *последовательные* операторы подобны операторам языков высокого уровня. На рис. 2.1 приведена общая структура VHDL-описания, из которой следует, что последовательные операторы (sequential statement) могут появляться внутри операторов процесса или внутри тел подпрограмм (функций, процедур).

На данном рисунке указаны основные параллельные и последовательные операторы.

Перечислим последовательные операторы:

1) оператор присвоения значения переменной;
2) оператор назначения сигнала, т.е. присвоения значения сигналу;
3) оператор **if** (если);
4) оператор **case** (случай);
5) оператор **loop** (цикл);
6) оператор **next** (следующий);
7) оператор **exit** (выход);
8) оператор **null** (нуль, пустой);
9) оператор вызова процедуры;
10) оператор **return** (возврат);
11) оператор **assert** (сообщение);
12) оператор **wait** (ожидать).

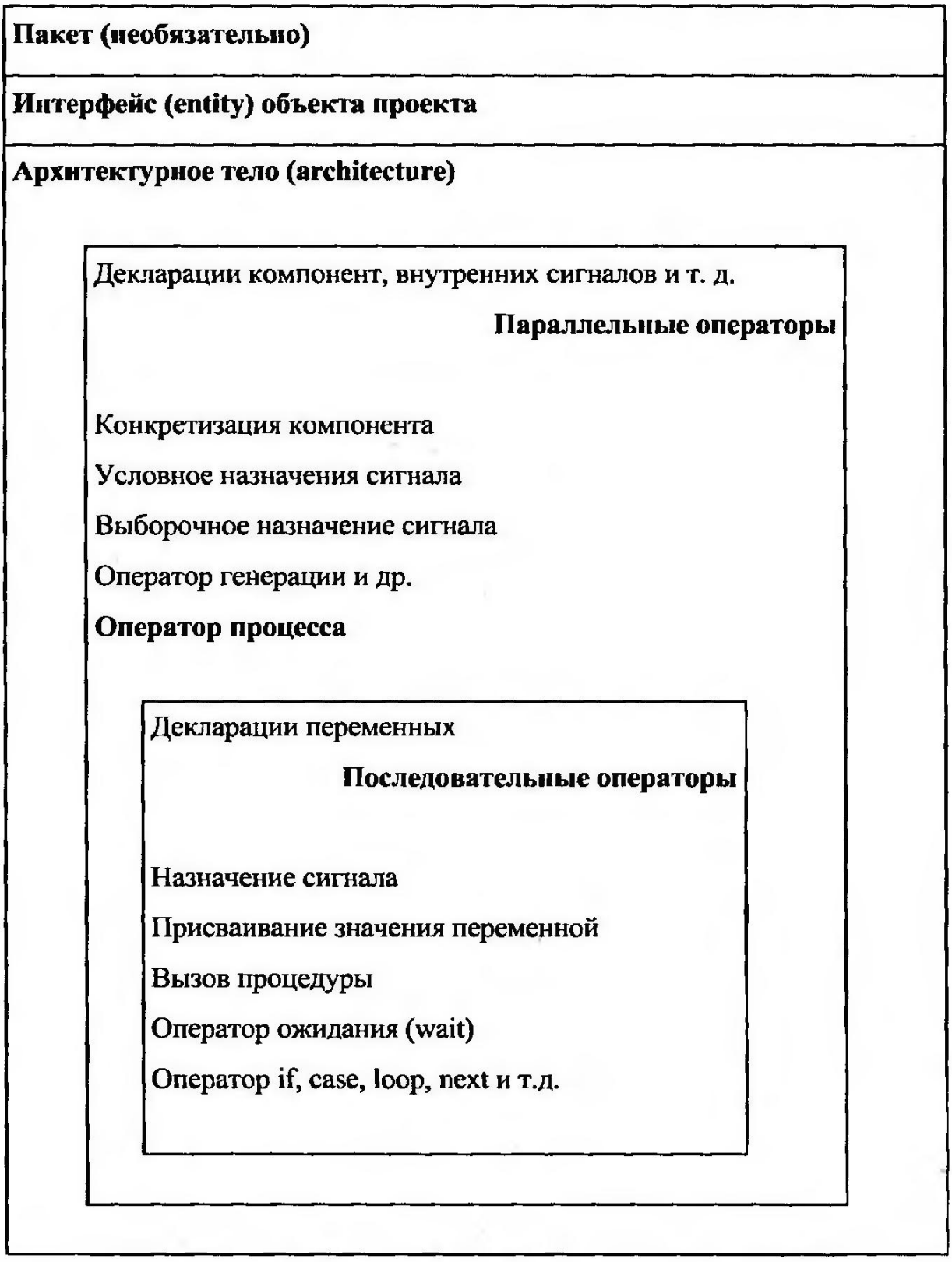

Рис. 2.1. Структура VHDL-описания

Оператор присваивания значения переменной

Определение.

variable_assignment_statement::=
[label] target := expression ;

Данный оператор заменяет текущее значение (target) переменной новым значением, которое определяется выражением (expression). Переменная и выражение должны быть того же базового типа.

Еще раз напомним, что присваивание значения переменным не есть то же самое, что сигналам. Присваивание значений сигналам мы обсудим в следующем разделе.

В VHDL локальные переменные могут быть только декларированы в области операторов процессов и подпрограмм (функций или процедур).

В следующем VHDL-коде приведены примеры присваивания значений переменным. Слева указаны номера строк, не относящиеся к тексту на языке VHDL.

```
1   entity VAR is
2   end VAR;
3
4   architecture functional of VAR is
5   signal A, B, J : bit_vector(1 downto 0);
6   signal E, F, G : bit;
7   begin
8   p0 : process (A, B, E, F, G, J)
9   variable C, D, H, Y : bit_vector(1 downto 0);
10   variable W, Q      : bit_vector(3 downto 0);
11   variable Z         : bit_vector(0 to 7);
12   variable X         : bit;
13   variable DATA   : bit_vector(31 downto 0);
14   begin
15   C       := "11";
```

```
16   X      := E and F;
17   Y      := H nand J;
18   Z(0 to 3) := C & D;                    -- конкатенация
19   Z(4 to 7) := (not A) & (A nor B);      -- конкатенация
20   D      := ('0', '0');                  -- агрегат
21   W  := (2 downto 1 => G, 3 => '1', others => '0');  -- агрегат
22   DATA     := (others => '1');                       -- агрегат
23   end process;
24   end functional;
```

В строке 15 переменная C получает константное значение. Выражения в строке 16, 17 используют логические операторы.

В строке 18 в выражении употребляется оператор & конкатенации, чтобы присвоить значения первым четырем битам переменной Z.

В строке 19 употреблена комбинация логических операторов и конкатенация.

Строка 20 показывает агрегат, в котором употребляется позиционное отображение (соответствие). Запись ('0', '0') называется *агрегатом*. Агрегат заключается в круглые скобки, входящие в агрегат элементы разделяются запятой.

В строке 21 употребляется позиционное отображение и ключевое слово **others**.

В строке 22 всем компонентам битового вектора — переменной DATA — присваивается значение единица.

Заметим, что локальные переменные «видны» только внутри процессов или подпрограмм, которые декларированы. VHDL'93 определяет другой класс переменных, называемых **shared** (совместно используемые, общие), которые могут совместно использоваться (видны) с процессами и подпрограммами. Понятие «видимости» будет рассмотрено далее.

Агрегаты и конкатенация могут использоваться не только при присвоении значений переменным, но и при назначении сигналов для таких типов данных, как массивы. Рассмотрим пример агрегата.

```
Variable z_bus      : bit_vector (3 downto 0 );
Variable  A,B,C,D  : bit;
          z_bus := (A,B,C,D);              -- агрегат
```

Запись (A,B,C,D) является агрегатом.

Присваивание значений сигналам (назначение сигналов)

В языке VHDL в операторах назначения сигналов, т. е. в операторах присваивания значений сигналам используются два вида задержек:

- *инерционная* задержка;
- *транспортная* задержка;

Ключевое слово **inertial** определяет инерционную задержку, ключевое слово **transport** определяет транспортную задержку.

Пример.

```
X<= inertial Y after 3 ns;        -- инерционная задержка
X<= transport Y after 3ns;        -- транспортная задержка.
```

В случае инерционной задержки передача сигнала будет иметь место, если и только если входной сигнал будет сохранять соответствующий уровень в течение заданного отрезка времени. В языке VHDL этот заданный отрезок времени и есть задержка, указываемая во фразе **after**. Таким образом, в первом примере изменение значения Y подействует на значение X только в случае, если новый уровень Y будет сохраняться в течение 3ns и более.

Во втором примере (транспортная задержка) все изменения Y будут передаваться в X независимо от того, сколько времени будет сохраняться новое значение Y.

! Если не используется ключевое слово **transport**, то подразумевается инерционная задержка.

Пример

```
X<= Y after 3 ns;                 -- инерционная задержка
```

Как отмечено в [1], механизм инерционной задержки позволяет отфильтровывать входные сигналы, которые меняются слишком быстро, т. е. если длительность сигнала Y меньше 3 ns, то его значение не будет передано сигналу X. Данный механизм по существу имитирует работу реальной схемы. Уровень напряжения определяет значение логического сигнала. Ввиду наличия электрических емкостей напряжения узлов не могут изменяться мгновенно, необходимо, чтобы определенное количество энергии подавалось в течение определенного отрезка времени, — только в этом случае напряжение узла изменится настолько, чтобы вызвать переключение схемы, управляемой этим напряжением. Поэтому при моделировании реальных логических схем используется инерционная задержка.

Транспортная задержка чаще используется на этапе алгоритмического проектирования.

Укажем **различия** между локальными переменными и сигналами.

1. Локальные переменные декларируются и видны только внутри процесса или подпрограммы. Сигналы не могут быть декларированы внутри процесса или подпрограммы.

2. Новое значение локальной переменной немедленно корректируется, когда выполняется оператор присваивания. Понятие времени не ассоциируется с понятием переменной. Оператор назначения сигнала корректирует сначала драйвер сигнала. Когда процесс станет приостановленным, сигнал корректируется. Поэтому использование сигналов ведет к двухпроходному моделированию. С точки зрения системы моделирования переменные применять «дешевле».

3. Только сигналы могут употребляться для связывания параллельных операторов.

! Порты, декларируемые в **entity,** являются сигналами. Аргументы подпрограмм могут быть сигналами или переменными.

4. В VHDL-описаниях логических (цифровых) схем сигнал употребляется для описания соединений элементов. Локальные пе-

ременные обычно употребляются как временные значения в алгоритме описания функции.

Пример. (Различие между локальной переменной и сигналом).

Данный VHDL-код

```
Y<= A + (B * C + D * E * F + G);
Z<= A – (B * C + D * E * F + G);
```

эквивалентен следующему VHDL-коду

```
V:=(B * C + D * E * F + G);
Y<= A + V;
Y<= A – V;
```

однако не эквивалентен приведенному ниже VHDL-коду

```
V<=(B * C + D * E * F + G);
Y<= A + V;
Z<= A – V;
```

Замечание. Запись x <= y <= z понимается не как «конвейерное» назначение сигналов. Правильное понимание: сигналу x присваивается значение, равное значению выражения y<=z (y меньше либо равно z).

Назначение сигналов в случае массивов, например битовых векторов, является *позиционным.*

Пример.

```
Signal z_bus : bit_vector (3 downto 0);
Signal c_bus : bit_vector (1 to 4);
```

z_bus <= c_bus;	эквивалентно	z_bus(3) <=c_bus(1);
		z_bus(2) <=c_bus(2);
		z_bus(1) <=c_bus(3);
		z_bus(0) <=c_bus(4);

При назначении сигналов должно указываться то направление диапазона (возрастающий диапазон — **to**, убывающий диапазон — **downto**), которое было при декларации массива.

Для предыдущего примера

```
z_bus (3 downto 2) <= "00"; -- правильно
c_bus (2 to 4) <= z_bus (3 downto 1); -- правильно
z_bus (0 to 1) <= "11"; -- неправильно, так как z_bus декларирован
                        -- с убывающим диапазоном
```

Сигналы и переменные одного и того же типа могут быть присвоены один другому.

Пример использования агрегатов при назначении сигналов.

```
Signal z_bus : bit_vector (3 downto 0 );
Signal A, B, C, D : bit;
  z_bus <= (3 => '1',  1 downto 0 => '1', 2 => B);  --  агрегат
```

***Оператор* if (*если*)**

Общий вид оператора if

```
if условие then
   упорядоченное множество последовательных операторов
         {elsif условие then
  упорядоченное множество последовательных операторов}
         [else
  упорядоченное множество последовательных операторов]
end if;
```

Оператор if языка VHDL подобен операторам **if** в других языках программирования.

Выражение, представляющее собой «условие» должно иметь тип BOOLEAN. В одном **if** операторе может быть одна (ни одной) либо более частей **elsif**. Ключевое слово **elsif** следует отличать от слов **else if**. Часть **else** может быть только одна (или ни одной). Должен быть разделитель между ключевыми словами в заключительной фразе **end if**;

Следующая модель 5-битового счетчика употребляет **if** операторы [9].

```
entity IFSTMT is
  port (
    RSTn, CLK, EN, PL : in  bit;
    DATA              : in  integer range 0 to 31;
    COUNT             : out integer range 0 to 31);
end IFSTMT;

architecture RTL of IFSTMT is
  signal COUNT_VALUE : integer range 0 to 31;
begin
  p0 : process (RSTn, CLK)
  begin
    if (RSTn = '0') then
      COUNT_VALUE <= 0;
    elsif (CLK'event and CLK = '1') then
      if (PL = '1') then
        COUNT_VALUE <= DATA;
      elsif (EN = '1') then
        if (COUNT_VALUE = 31) then
          COUNT_VALUE <= 0;
        else
          COUNT_VALUE <= COUNT_VALUE + 1;
        end if;
      end if;
    end if;
  end process;
  COUNT <= COUNT_VALUE;
end RTL;
```

Как показано в разделе деклараций, 5-битовый счетчик имеет порты RSTn, CLK, EN, PL, DATA. Выходной порт COUNT получает

значение счетчика, RSTn — асинхронная установка (в нуль), CLK — входной сигнал синхронизации, PL — параллельное считывание, DATA — порт данных.

***Оператор* case (случай)**
Общий вид оператора case

```
case выражение is
when выбор => упорядоченное множество
                    последовательных операторов
  {when выбор => упорядоченное множество
                    последовательных операторов }
end case;
```

Оператор **case** выбирает одну из альтернатив, избранная альтернатива (случай) определяется значением выражения. Выражение должно быть *дискретного* типа или типа одноразмерного массива символов, значения которых могут быть представлены как строки или строка битов. Выбор должен быть такого же типа, как выражение. Все возможные выборы (случаи) должны быть перебраны. Для случая **«others»** (другие) должно быть такое значение, которое не соответствует предыдущим альтернативам.

Оператор **case** является подходящим для моделирования конечных автоматов и программ микропроцессора. Используя оператор **case,** приведем пример [9] VHDL-кода для вычисления числа дней в каждом месяце.

```
package PACK is
  type month_type is (JAN, FEB, MAR, APR, MAY, JUN,
                      JUL, AUG, SEP, OCT, NOV, DEC);
end PACK;

use work.PACK.all;
entity CASESTMT is
  port (
    MONTH   : in  month_type;
    LEAP    : in  boolean;
```

```
      DAYS    : out integer);
  end CASESTMT;

  architecture RTL of CASESTMT is
  begin
    p0 : process (LEAP, MONTH)
    begin
      case MONTH is
        when FEB =>
          if LEAP then
            DAYS <= 29;
          else
            DAYS <= 28;
          end if;
        when APR | JUN | SEP | NOV =>
          DAYS <= 30;
        when JUL to AUG =>
          DAYS <= 31;
        when others =>
          DAYS <= 31;
      end case;
    end process;
  end RTL;
```

***Оператор* loop (цикл)**

Общий вид оператора loop

```
[метка цикла:][while условие |
        for идентификатор in диапазон дискретного типа]
        loop
        упорядоченное множество последовательных операторов
end loop [метка цикла];
```

Когда в записи цикла используется ключевое слово **while**, то сначала вычисляется условие (**condition**). Если условие есть TRUE,

выполняется последовательность последовательных операторов, иначе оператор цикла завершается.

Когда в записи цикла используется ключевое слово **for**, то идентификатор определяет цикловой параметр (счетчик числа итераций цикла) с базовым дискретным типом. Параметр цикла в этом случае *не требуется декларировать* снаружи цикла, он употребляется как константа внутри действия оператора цикла и он не может быть целью оператора присваивания.

! Общая ошибка: употребление параметра цикла снаружи оператора цикла.

Пример.

```
loop1: for i in 0 to 9 loop exit loop1 when A(i) > 20;
next when A(i) > 10; -- цикловой параметр i не требует
sum:= sum + A(i);     -- предварительной декларации
end loop loop1;

if i = 20 then      -- ошибка! Параметр цикла снаружи цикла.
```

***Оператор* next (следующий)**
Общий вид оператора next

next [метка цикла][**when** условие];

Пример дан выше. Оператор **next** употребляется для перехода к следующей итерации цикла.

***Оператор* exit (выход)**
Общий вид оператора exit

exit [метка цикла] [**when** условие];

Оператор EXIT употребляется, чтобы завершить выполнение и закрыть оператор цикла. Если условие (condition) есть TRUE, то осуществляется выход из цикла.

***Оператор* null (пустой)**

Общий вид оператора null (нуль, пустой)

null;

Оператор **null** не представляет действий. Он употребляется, чтобы точно специфицировать, что нет действий. Типичное применение — в операторе case, чтобы определить действия во всех случаях.

***Оператор* вызова процедуры**

Оператор вызова процедуры состоит из имени процедуры с аргументами (если они есть) в скобках. Приведем пример определения и вызова функции и процедуры.

```
entity CALL_PRO is
end CALL_PRO;

architecture RTL of CALL_PRO is

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;
procedure left_one (
signal DATA : in bit_vector (1 to 8);
signal l_bit : out integer) is
variable temp : integer;
begin
temp := 0;
for i in 1 to 8 loop
if (DATA(i) = '1') then
temp := i;
end if;
if (temp /= 0) then exit;
```

```
end if;
end loop;
l_bit <= temp;
end left_one;

signal DIN : bit_vector (1 to 8);
signal bit_1 : bit;
signal bool_1 : boolean;
signal DOUT : integer;
begin
p0: process (bit_1,DIN)
begin
bool_1 <= bit_bool(bit_1);        -- вызов функции
LEFT_ONE(DIN, DOUT);              -- вызов процедуры
end process;
p1: process
begin
bit_1 <= '1' after 20 ns, '0' after 40 ns;
DIN <= "01010000" after 20 ns,
       "00000000" after 40 ns,
       "00001100" after 60 ns,
       "00000001" after 80 ns;
 wait for 100 ns;
end process;

end RTL;
```

Функция bit_bool преобразует тип BIT в тип BOOLEAN. Предлагаем читателю провести моделирование, проанализировать временную диаграмму и определить функциональнос назначение процедуры left_one.

***Оператор* return (возврат)**

Общий вид оператора return

```
return [выражение];
```

Употребляется, чтобы завершить выполнение самой внутренней функции или процедуры. Он используется только внутри тела функции или процедуры. Оператор **return** не требуется в теле процедуры, поэтому в архитектурном теле RTL (entity RETURNSTMT) соответствующая строка может быть удалена. Оператор **return** может быть употреблен с другими последовательными операторами, такими как **if**, **case** для управления возвратом функции или процедуры.

***Оператор* assert (сообщение)**
Общий вид оператора assert

assert условие [**report** выражение] [**severity** выражение];

Операторы сообщений проверяют, является ли условие истинным (**TRUE**), и сообщают об ошибке, если условие является ложным. По умолчанию сообщенное выражение есть **«Assertion violation»** (нарушение). Выражение с ключевым словом **severity** (severity — степень серьезности) имеет перечислимый тип: **NOTE, WARNING, ERROR, FAILURE**.

Примеры.

assert (CLK'event **and** CLK='0') **report** "D hold error" **severity** WARNING;

assert (CLK'last_event > HOLD) **report** "D hold error" **severity** ERROR;

В данных примерах атрибут CLK'last_event имеет тип TIME и возвращает время, пройденное с момента последнего изменения сигнала CLK, HOLD — имеет тип TIME. Условие CLK'last_event>HOLD может быть либо истинным, либо ложным.

Следующие два оператора (две строки) эквивалентны:

report "NEW YEAR 3003" **severity** ERROR;
assert FALSE **report** "NEW YEAR 3003" **severity** ERROR;

также, как и следующие:
report "NEW YEAR 2002";
assert FALSE **report** "NEW YEAR 2002" **severity** NOTE;

Так как FALSE всегда является ложным, то данные сообщения всегда выдаются — это примеры безусловно выдаваемых сообщений.

***Оператор* wait (ожидать)**
Общий вид оператора wait

wait on список чувствительности **until** условие **for** тайм-аут ;

Оператор **wait** является причиной временного прекращения оператора процесса или процедуры. Оператор ожидания **wait** приостанавливает процесс до момента, пока не изменится некоторый сигнал в списке чувствительности процесса, в это время будет произведено вычисление условия. Фраза «условие» есть выражение типа BOOLEAN. Если получается истинное значение, выполнение процесса возобновляется. Фраза «тайм-аут» устанавливает максимальное время ожидания, после которого процесс возобновит свое выполнение.

Пример.

WAIT on A, B **until** (C = 0) **for** 50 ns;

Этот оператор приостановит процесс до момента изменения A или B, после чего будет проверено выражение C = 0 и, если результатом проверки будет истина, процесс возобновится. Но независимо от этих условий возобновление процесса произойдет через 50ns.

Допустимо записывать одно или более условий в операторе ожидания, например,

Условие 1.	**WAIT on** A, B;
Условие 2.	**WAIT until** (C = 0);
Условие 3.	**WAIT for** 50 ns;

В условии 1 процесс будет возобновляться, когда изменится A или B.

В условии 2 нет списка сигналов запуска, поэтому процесс возобновится, когда C изменит свое значение из 1 в 0.

В условии 3 процесс возобновится через 50ns независимо от любых других условий.

В цифровых системах логические процессы часто приостанавливаются в своем выполнении, ожидая истечения некоторого периода времени или наступления некоторого события. После истечения указанного периода времени или наступления ожидаемого события выполнение процесса возобновляется. Эта ситуация иллюстрируется следующим образом

```
process -----   -- начало выполнения процесса
----------
----------
----------
wait            -- оператор ожидания
----------      -- возобновление выполнения процесса
----------
----------
end process;    -- конец выполнения процесса
```

! Для оператора процесса мы можем иметь либо список чувствительности после ключевого слова процесс, либо оператор wait, но не оба вместе. Может быть болес одного оператора wait внутри оператора процесса.

Примеры оператора wait.

1. Оператор wait типа **for**

```
wait for 10ns;
wait for CLK_Period/2;
```

2. Оператор wait типа **until**

```
wait until CLK='1';
wait until CE and (not RST);
wait until IntData>16;
```

3. Оператор wait типа **on**

wait on CLK;
wait on Enable, Data;

wait until Enable ='1';	эквивалентно	**loop** **wait on** Enable; **exit when** Enable = '1'; **end loop;**

4. Комбинированный оператор wait (комбинация двух или трех предыдущих)

wait on Data **until** CLK= '1';
wait until CLK= '1' **for** 10ns;

2.2. Параллельные операторы

Параллельные операторы в VHDL определяют параллельное (во времени) поведение схем.

! Порядок выполнения параллельных операторов не связан с порядком их появления внутри архитектурного тела.

Параллельные операторы активизируются сигналами, которые употребляются для связи параллельных операторов. Последовательные операторы выполняются в порядке их появления в VHDL-коде.

Перечислим параллельные операторы:

1) оператор **process** (процесс);
2) оператор параллельного сообщения;
3) оператор параллельного вызова процедуры;
4) оператор условного назначения сигнала;
5) оператор **select** выборочного назначения сигнала;
6) оператор конкретизации (создания экземпляра) компонента;

7) оператор **generate** (генерации);
8) оператор **block** (блок).

Параллельный оператор **process**
Общий вид оператора process (процесс)

[имя процесса :] [**postponed**] **process** [(список чувствительности)]
раздел деклараций
begin
упорядоченное множество последовательных операторов
end process [имя процесса];

Оператор процесса есть параллельный оператор, который определяет независимое последовательное поведение некоторой части проекта, описанное упорядоченной совокупностью последовательных операторов. Метка процесса необязательна, однако если она есть в конце (после слов **end process**), то она должна быть и вначале перед словом **process.**

! Ожидание (wait) в начале процесса не эквивалентно ожиданию в списке чувствительности процесса.

process(signal_1) statement_1; statement_2; . . . statement_n; end process;	эквивалентно	process statement_1; statement_2; . . . statement_n; wait on signal_1; end process;
process(signal_1) statement_1; statement_2; . . . statement_n; end process;	не эквивалентно	process wait on signal_1; statement_1; statement_2; . . . statement_n; end process;

! Процесс может иметь список сигналов запуска и один (или более) операторов ожидания, но не оба вместе.

! Сигналы не могут быть декларированы внутри процессов.

В *декларативной части* процесса могут быть: тела подпрограмм; декларации подтипов; декларация констант; декларация файлов; декларация атрибутов; спецификации атрибутов и др.

Ключевое слово postponed характеризует отложенный процесс. Если имеется несколько параллельных процессов и среди них отложенный, то отложенный процесс активизируется в течение только *последней* дельта-задержки временной точки [7].

Так как в теле оператора process располагаются (и выполняются строго по порядку при моделировании) именно последовательные операторы, то для промежуточных результатов вычислений в процессах рекомендуется использовать *переменные*, а не сигналы, в противном случае требуемое поведение будет запаздывать по времени [1].

Оператор параллельного сообщения

Синтаксис *оператора параллельного сообщения* **(assert)** такой же, как и у оператора последовательного сообщения.

«Параллельность» заключается в том, что оператор **assert** может присутствовать в параллельных процессах.

Оператор параллельного вызова процедуры

Общий вид оператора

[метка:] оператор вызова процедуры

Оператор параллельного вызова процедуры представляет процесс, содержащий оператор последовательного вызова процедуры. Его выполнение эквивалентно оператору процесса. В примере показано, что процедуры могут вызываться параллельно.

```
1   entity call_parallel is
2     port (
```

```
3       data_inp  : in  bit_vector(5 downto 0);
4       data_out : out bit_vector(1 downto 0));
5   end call_parallel;
6   architecture RTL of call_parallel is
7     procedure N_XOR (
8       signal x1, x2, x3 : in bit;
9       signal f          : out bit) is
10     begin
11       f <= x1 xor x2  xor x3;
12     end N_XOR;
13   begin
14     N_XOR (x1 => data_inp(5), x2 => data_inp(4), x3 =>
data_inp(3),
15           f => data_out(1));
16     p0 : N_XOR (data_inp(2), data_inp(1), data_inp(0), data_out(0));
17   end RTL;
```

В строках 14, 16 осуществляется параллельный вызов процедур. В строке 16 параметры передаются позиционным сопоставлением, в строке 14 передача параметров осуществляется сопоставлением имен: вместо параметра x1 передается data_inp(5), вместо x2 передается data_inp(4) и т. д.

! Каждый формальный параметр процедуры должен быть типа константы или сигнала.

Декларация процедур и тела процедур будут обсуждены далее.

Параллельный оператор условного назначения сигнала

Общий вид оператора

[guarded][transport]
{назначение сигнала **when** условие **else**} сигнал;

Оператор параллельного назначения сигнала эквивалентен оператору процесса, назначающему значения сигналам. Могут быть употреблены опции **guarded** (охраняемый) **transport** (транспортный).

Пример, показывающий эквивалентность параллельного оператора условного назначения сигнала (архитектурное тело first) и оператора процесса (архитектурное тело second).

```
entity example_condition is
  port (
    x1, x2, x3, x4 : in bit;
   condition       : in bit_vector(1 downto 0);
    F              : out bit);
end example_condition;
architecture first of example_condition is
begin
  F <= x1 when condition = "00" else
       x2 when condition = "01" else
       x3 when condition = "10" else
       x4;
end first;
architecture second of example_condition is
begin
  process (x1, x2, x3, x4, condition )
  begin
    if (condition = "00") then
      F <= x1;
    elsif (condition = "01") then
      F <= x2;
    elsif (condition = "10") then
      F <= x3;
    else
      F <= x4;
    end if;
  end process;
end second;
```

В примере все условия (condition) являются различными, поэтому сигнал F получает одно из значений из множества {x1, x2, x3, x4}. Какое же значение получит сигнал, если в нескольких местах будет удовлетворяться условие? Ответ следующий: будет выполняться назначение для первого выполненного условия.

Параллельный оператор* select *выборочного назначения сигнала

Общий вид оператора **select** (выбирать)

with выражение **select**
имя сигнала <= [**guarded**][**transport**]
{имя сигнала **when** условие выбора, }
имя сигнала условие выбора;

Данный оператор эквивалентен оператору процесса.

Пример, показывающий эквивалентность параллельного оператора выборочного назначения сигнала (архитектурное тело first) и оператора процесса (архитектурное тело second).

```
entity example_selection is
  port (      x1, x2, x3, x4 : in  bit;
                  selection : in  bit_vector(1 downto 0);
                          F : out bit);
end example_selection;
architecture first of example_selection is
begin
  with selection select
  F <= x1 when "00",
       x2 when "01",
       x3 when "10",
       x4 when others;
end first;

architecture second of example_selection is
begin
  process (x1, x2, x3, x4, selection)
```

```
  begin
    case selection is
      when "00" => F <= x1;
      when "01" => F <= x2;
      when "10" => F <= x3;
      when others => F <= x4;
    end case;
  end process;
end second;
```

Параллельный оператор конкретизации компонента

Общий вид оператора конкретизации компонента (оператора создания экземпляра компонента)

метка : имя компонента
[**generic** (список параметров);]
[**port map** (список портов)];

Этот оператор употребляется для структурной организации проекта. Часть схемы (подсхема) описывается как компонент (component), имеющий имя (name). Одна и та же подсхема может входить в схему несколько раз, однако при этом она имеет различные связи. Чтобы описать эти связи, употребляется *оператор создания экземпляра компонента* (*оператор конкретизации компонента*), т. е. имеется в виду конкретизация связей данной подсхемы.

Соответствие портов при создании экземпляров компонентов может быть осуществлено

- позиционным сопоставлением;
- ключевым соответствием, с использованием оператора «=>».

Пример ключевого соответствия портов при создании экземпляров компонентов.

```
p1: A port map (x1 => x1, y1 => w) ;  -- ключевое соответствие
p2: B port map (x2 => w, y2 => z) ;   -- ключевое соответствие
p3: C port map (y3 => v, x3 => z) ;   -- ключевое соответствие
```

Сделаем пояснения. При создании экземпляра компонента A портам x1, y1 компонента A ставятся в соответствие сигналы x1, w. При создании экземпляра компонента B портам x2, y2 компонента B ставятся в соответствие сигналы w, z (рис. 2.2). При ключевом соответствии порядок перечисления портов не играет роли, поэтому при ключевом соответствии портов можно сначала указать выходной порт, как это сделано при создании экземпляра компонента C.

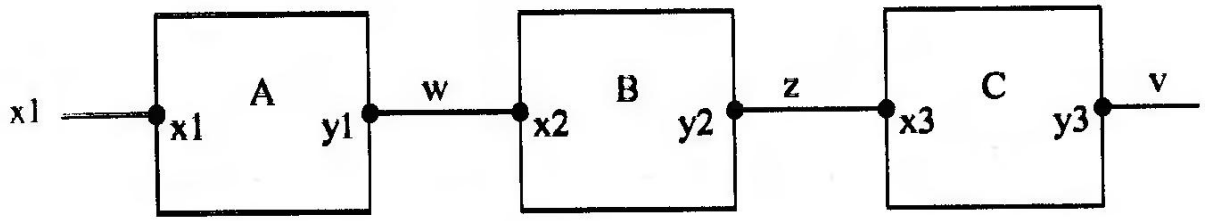

Рис. 2.2. Конкретизация (создание экземпляров) компонентов

Пример позиционного сопоставления (соответствия) портов при создании экземпляров компонентов (рис. 2.2).

```
p1: A port map(x1, w); -- позиционное соответствие
p2: B port map(w, z); -- позиционное соответствие
p3: C port map(z, v); -- позиционное соответствие
```

Рассмотрим 8-разрядный сдвиговый регистр (рис. 2.3), в состав которого входит восемь подсхем — D-триггеров (элементов памяти) [9]. D-триггер имеет имя DFF.

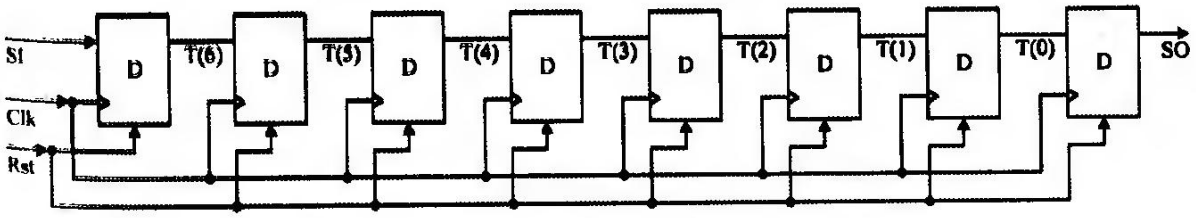

Рис. 2.3. Сдвиговый регистр

```
1   entity DFF is
2     port (
3       RSTn, CLK, D : in bit;
4       Q          : out bit);
5   end DFF;
6   architecture RTL of DFF is
7   begin
8     process (RSTn, CLK)
9     begin
10      if (RSTn = '0') then
11        Q <= '0';
12      elsif (CLK'event and CLK = '1') then
13        Q <= D;
14      end if;
15    end process;
16  end RTL;
17  ----------------------------------------
18  entity SHIFT is
19    port (
20      RSTn, CLK, SI : in  bit;
21      SO          : out bit);
22  end SHIFT;
23  architecture RTL1 of SHIFT is
24    component DFF
25    port (
26      RSTn, CLK, D : in  bit;
27      Q          : out bit);
28    end component;
29    signal T : bit_vector(6 downto 0);
30  begin
31    bit7 : DFF
32   port map (RSTn => RSTn, CLK => CLK, D => SI, Q => T(6));
33    bit6 : DFF
34      port map (RSTn, CLK, T(6), T(5));
```

```
35      bit5 : DFF
36        port map (RSTn, CLK, T(5), T(4));
37      bit4 : DFF
38  port map (CLK => CLK, RSTn => RSTn, D => T(4), Q => T(3));
39      bit3 : DFF
40        port map (RSTn, CLK, T(3), T(2));
41      bit2 : DFF
42        port map (RSTn, CLK, T(2), T(1));
43      bit1 : DFF
44        port map (RSTn, CLK, T(1), T(0));
45      bit0 : DFF
46        port map (RSTn, CLK, T(0), SO);
47    end RTL1;
```

Схема SHIFT задает сдвиговый регистр — каскадное соединение D-триггеров (элементов памяти). D-триггер описан в строках 1–16.

Сдвиговый 8-битовый регистр специфицирован в строках 18–22.

Строки 24–28 декларируют компонент DFF.

Строка 29 декларирует 7-битовый сигнал T, употребляемый для связи между соседними триггерами.

Компонент DFF конкретизирован (упомянут) восемь раз, чтобы получить сдвиговый регистр.

! Каждый оператор создания экземпляра компонента должен иметь метку. Метки играют роль имен элементов схемы.

Карта портов дается в скобках после ключевых слов **port map**.

Назначение портов компонентов является как *позиционным* (см. метки bit0, bit1, bit2, bit3, bit5, bit6), так и *ключевым* (см. метки bit4, bit7).

! Для выходных неиспользуемых портов компонентов нужно употребить ключевое слово **open**.

Следующий фрагмент VHDL-кода показывает, что при создании экземпляра компонента add1 на вход b1 можно подать константу 0 и выход c1 не использовать.

P1: add1 **port map** (b1 => '0', b2 =>x, s1 => s1, c1 => **open**);

Рассмотрим еще один пример — 7-разрядный сумматор, являющийся каскадным соединением одного полусумматора add1 и шести одноразрядных сумматоров add2. Схема данного сумматора изображена на рис. 2.4,а.

```
entity adder_N_comp is
port (a, b : in bit_vector (0 to 6);
      s : out bit_vector (0 to 6);
      c  : out bit);
end adder_N_comp;
architecture structural of adder_N_comp is
component
add1
port (b1, b2: in BIT;
    c1, s1: out BIT);
end component;
component  add2
port(c1, a1, a2:in BIT;
    c2, s2:out BIT);
end component;
signal c_in : bit_vector (0 to 5);

begin
p0: add1
port map (b1 => a(0), b2 => b(0), c1 => c_in(0), s1 => s(0) );
p1: add2
port map (c1=> c_in(0), a1 => a(1), a2 => b(1), c2 => c_in(1),
s2 => s(1));
p2: add2
port map (c1 => c_in(1), a1 => a(2), a2 => b(2), c2 => c_in(2),
s2 => s(2));
p3: add2
```

```
        port map (c1 => c_in(2), a1 => a(3), a2 => b(3), c2 => c_in(3),
s2 => s(3));
        p4: add2
        port map (c1 => c_in(3), a1 => a(4), a2 => b(4), c2 => c_in(4),
s2 => s(4));
        p5: add2
        port map (c1 => c_in(4), a1 => a(5), a2 => b(5), c2 => c_in(5),
s2 => s(5));
        p6: add2
        port map (c1 => c_in(5), a1 => a(6), a2 => b(6), c2 => c,
s2 => s(6));
        end structural;
```

Примеры, приведенные для пояснения оператора создания экземпляров компонентов, понадобятся нам в дальнейшем. Схемы сдвигового регистра и сумматора являются регулярными и могут быть описаны более компактно с помощью оператора генерации, рассматриваемого далее.

Параллельный оператор **generate**

Общий вид оператора generate (генерации)

```
        метка : for параметр генерации generate
                    if условие  generate
                параллельные операторы
        end generate [метка];
```

Параметр генерации — константа дискретного типа в определенном диапазоне. Параметром генерации не может быть декларированная переменная или сигнал.

Оператор генерации позволяет сокращенно (по существу используя цикл) описывать совокупности повторяющихся операторов, в том числе и операторов конкретизации компонентов, т. е. оператор генерации представляет собой механизм для проектирования (описания) регулярных (систолических) структур.

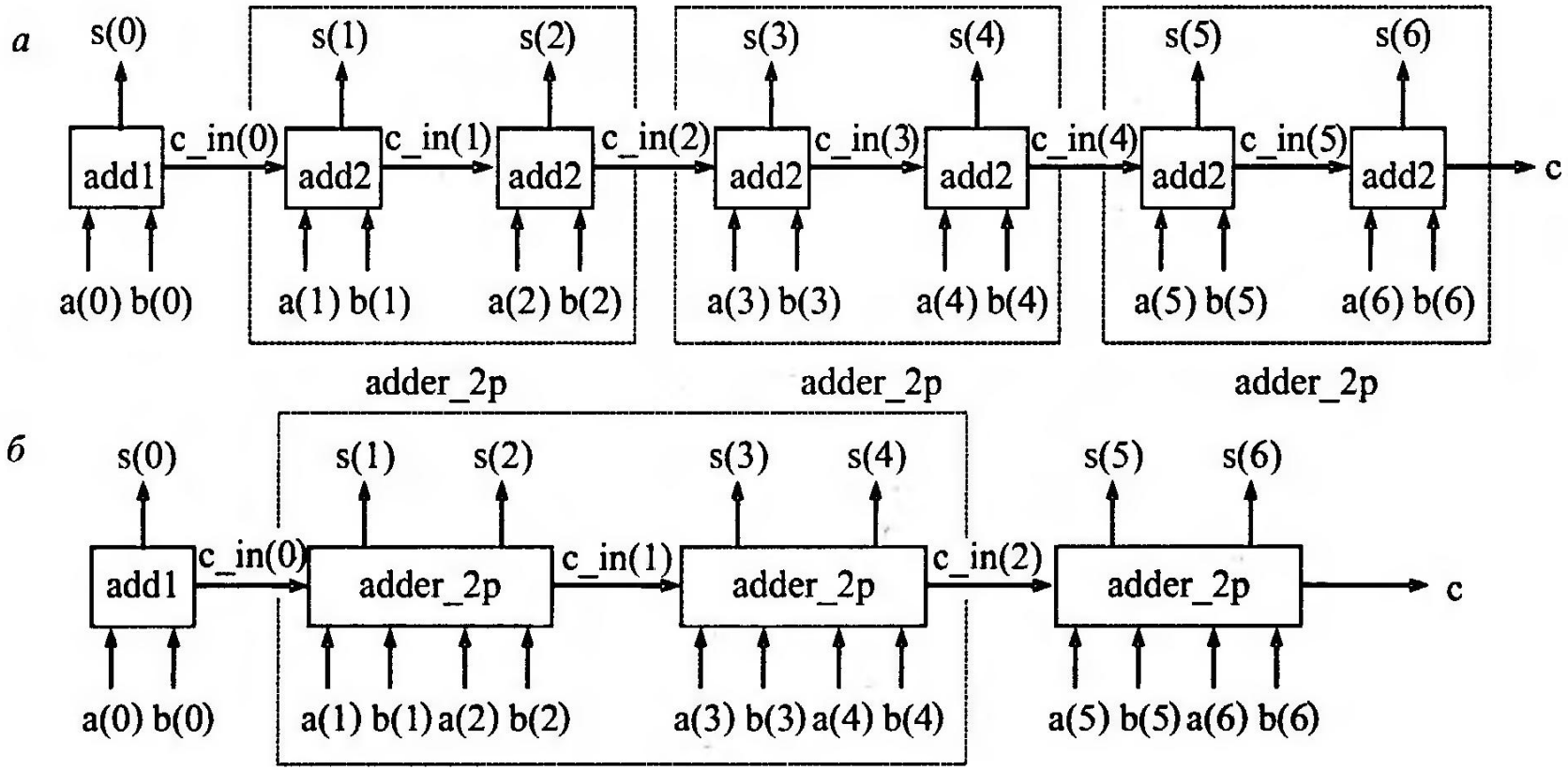

Рис. 2.4. 7-разрядный сумматор: а — в виде каскадного соединения одноразрядного полусумматора add1 и одноразрядных сумматоров add2; б — в виде каскадного соединения одноразрядного полусумматора add1 и двухразрядных сумматоров adder_2p

Структура регистра регулярна. Мы можем создать N экземпляров компонента DFF (D-триггера) и сделать N-битный сдвиговый регистр. Когда число N большое, значительно возрастает длина VHDL-кода. Оператор генерации представляет собой механизм для проектирования (описания) регулярных (систолических) структур.

Следующий пример [9] показывает применение оператора генерации к описанию 8-битового сдвигового регистра.

```
architecture RTL2 of SHIFT is
  component DFF
  port (
    RSTn, CLK, D : in bit;
    Q            : out bit);
  end component;
  signal T : bit_vector(6 downto 0);
begin
  g0 : for i in 7 downto 0 generate
    g1 : if (i = 7) generate
      bit7 : DFF
     port map (RSTn => RSTn, CLK => CLK, D => SI,
Q => T(6));
    end generate;
    g2 : if (i > 0) and (i < 7) generate
      bitm : DFF
        port map (RSTn, CLK, T(i), T(i - 1));
    end generate;
    g3 : if (i = 0) generate
      bit0 : DFF
        port map (RSTn, CLK, T(0), SO);
    end generate;
  end generate;
end RTL2;
```

Проанализировав данное описание, можно заметить, что некоторое «неудобство», связанное с установлением связей схемы в це-

лом с полюсами элементов, возникает при описании входного (метка bit7) и выходного триггера (метка bit0). Избежать этого неудобства можно, если увеличить размерность сигнала T .

```
architecture RTL3 of SHIFT is
  component DFF
  port (
    RSTn, CLK, D : in  bit;
    Q            : out bit);
  end component;
signal T : bit_vector(8 downto 0);      -- декларация сигнала T
begin
  T(8) <= SI;
  SO   <= T(0);
  g0 : for i in 7 downto 0 generate
    allbit : DFF
  port map (RSTn => RSTn, CLK => CLK, D => T(i + 1),
Q => T(i));
  end generate;
end RTL3;
```

Имеются два способа употребления оператора генерации.

Способ 1 — (способ **for**), синтаксис такой же, как у последовательного оператора **for loop**.

Способ 2 — (способ **if**) употребление подобно по синтаксису последовательному оператору **if**.

В архитектуре RTL2 способ **if** употреблен внутри способа **for**.

Укажем различия параллельного оператора генерации от последовательных операторов **for, if.**

1. Оператор генерации есть параллельный оператор, а **if, for loop** есть последовательные операторы.

2. Оператор генерации не имеет фраз **else, elsif.**

3. Необходима метка для оператора генерации.

4. Только параллельные операторы могут появляться внутри оператора генерации. Только последовательные операторы могут

появляться внутри последовательного **for loop** оператора и последовательного **if** оператора.

Архитектура DFF может быть декларирована в пакете и употреблена в архитектурах RTL1, RTL2, RTL3. Затем эта компонента может быть удалена из этих архитектур.

В архитектуре RTL3 параметр **i** не нуждается в декларации. Все три архитектуры RTL1, RTL2, RTL3 описывают то же самое поведение.

В следующем примере VHDL-кода оператор **generate** употребляется для спецификации N-разрядного сумматора adder_N (рис.2.5).

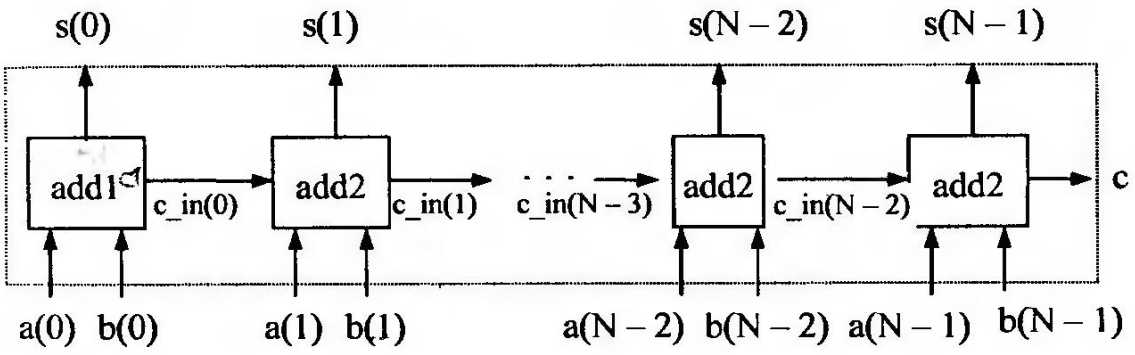

Рис. 2.5. N-разрядный сумматор

```
entity adder_N is
generic (N : natural := 4);
port (a, b : in bit_vector (0 to N – 1);
      s : out bit_vector (0 to N – 1);
      c  : out bit);
end adder_N;
architecture functional of adder_N is
component
add1
port (b1, b2: in BIT;
```

```
    c1, s1: out BIT);
end component;
component add2
port(c1, a1, a2:in BIT;
    c2, s2:out BIT);
end component;
signal c_in : bit_vector (0 to N – 1);
begin
adder: for i in 0 to N – 1 generate
first_bit: if (i = 0) generate
      first_cell:
      add1 port map (b1 => a(0), b2 => b(0),
                  c1 => c_in(0), s1 => s(0) );
end generate first_bit;

middle_bit: if (i>0) and (i < N – 1) generate
      middle_cell:
      add2 port map (c1 => c_in(i – 1), a1 => a(i), a2 => b(i),
                  c2 => c_in(i), s2 => s(i) );
end generate middle_bit;

end_bit: if (i = N – 1) generate
      end_cell:
      add2 port map (c1 => c_in(i – 1), a1 => a(i), a2 => b(i),
                  c2 => c, s2 => s(i) );
end generate end_bit;
end generate adder;
end functional;
```

Изменяя число N в строке с ключевым словом generic, можно получать описание сумматора требуемой разрядности. В представленном ниже архитектурном теле func_1 для N-разрядного сумматора используется только подсхема одноразрядного сумматора add2, при этом на вход переноса c1 подсхемы add2 (в разряде

с номером 0) подается нулевой сигнал, описание становится более компактным.

```
architecture func_1 of adder_N is
component  add2
port(c1, a1, a2:in BIT;
    c2, s2:out BIT);
end component;
signal c_in : bit_vector (0 to N);

begin       -- в схеме (рис. 2.5) все одноразрядные сумматоры — add2
c_in(0) <= '0';
adder: for i in 0 to N – 1 generate
i_bit_slice: add2 port map (c1 => c_in(i), a1 => a(i), a2 => b(i),
                            c2 => c_in(i + 1), s2 => s(i) );
end generate adder;
c <= c_in(N);
end func_1;
```

***Параллельный оператор* block**

Общий вид оператора block (блок)

```
имя блока (метка): block[(охранное выражение)]
                       заголовок блока
                       раздел деклараций
                   begin
              параллельные операторы
                   end block[имя блока];
```

Оператор блока определяет часть проекта (часть VHDL-описания цифровой системы, схемы). Напомним, что ***блок*** — это ограниченный фрагмент VHDL-кода, содержащий раздел описания и исполняемый раздел.

Блоки могут быть иерархически вложены и поддерживать тем самым декомпозицию проекта.

! Метка необходима в операторе блока.

В разделе деклараций блока размещаются

- декларации подпрограмм;
- тела подпрограмм;
- типы, подтипы;
- константы;
- сигналы;
- альтернативные точки входа в подпрограмму;
- декларации атрибутов;
- декларации констант;
- спецификации атрибутов;
- конфигурации.

Параллельные операторы размещаются в теле блока.

Необязательные охранные выражения будут обсуждены позднее.

Оператор блока обсудим на примере 7-разрядного сумматора, составленного из одного полусумматора и трех полных двухразрядных сумматоров.

Полный двухразрядный сумматор adder_2p есть каскадное соединение двух полных одноразрядных сумматоров add2.

Структурное описание схемы adder_2p приведено ниже, в данном описании a1, b1 — младшие разряды двухразрядных складываемых чисел **a** = (a2, a1), **b** = (b2, b1); a2, b2 — старшие разряды; c0 — перенос из предыдущего разряда. Таким образом, схема adder_2p реализует операцию сложения

$$(c0) + (a2, a1) + (b2, b1) = (c2, s2, s1)$$

одноразрядного числа c0 с двухразрядными числами **a, b.**

```
entity adder_2p is
port (a1, b1, a2, b2, c0 : in BIT;
      c2, s2, s1 : out BIT);
end adder_2p;
```

```
architecture structural of adder_2p is
component  add2
port(c1, a1, a2:in BIT;
    c2, s2:out BIT);
end component;
signal c1:BIT;
begin
circ1: add2
port map (c1 => c0, a1 => b1, a2 => b2, c2 => c1, s2 => s1);
circ2: add2
port map (c1 => c1, a1 => a1, a2 => a2, c2 =>c2, s2 => s2);
end structural;
```

В приводимом ниже описании две схемы adder_2p, предназначенные для сложения разрядов 1–4, описываются в виде блока. Подсхемы, входящие в блок, отмечены на рис. 2.4,б штриховой линией.

```
entity adder_N_block is
port (a, b : in bit_vector (0 to 6);
      s : out bit_vector (0 to 6);
      c  : out bit);
end adder_N_block;
architecture structural of adder_N_block is
component    add1
port (b1, b2: in BIT;
    c1, s1: out BIT);
end component;
component  adder_2p
port(a1, b1, a2, b2, c0:in BIT;
    c2, s2, s1:out BIT);
end component;
signal c_in : bit_vector (0 to 2);
begin
p0: add1 port map (b1 => a(0), b2 => b(0),
                   c1 => c_in(0), s1 => s(0) );
```

```
block0:block
begin
stage1: adder_2p port map (c0 => c_in(0), a1 => a(1), b1 => b(1),
a2 => a(2), b2 => b(2), c2 => c_in(1), s2 => s(2), s1 => s(1) );
stage2: adder_2p port map (c0 => c_in(1), a1 => a(3), b1 => b(3),
a2 => a(4), b2 => b(4), c2 => c_in(2), s2 => s(4), s1 => s(3) );
end block;
stage3: adder_2p port map (c0 => c_in(2), a1 => a(5), b1 => b(5),
a2 => a(6), b2 => b(6), c2 => c, s2 => s(6), s1 => s(5) );
end structural;
```

В данном примере отсутствуют охранные выражения, заголовок блока, раздел деклараций блока.

Блоки могут быть вложены.

```
architecture XX of SYSTEM is
- - - - -- раздел описаний внешнего блока
- - - -
- - - -
begin -- выполнимые операторы внешнего блока
- - - -
- - - -
A:    block
      - - - -      -- раздел описаний внутреннего блока A
      - - - -
      begin        -- выполнимые операторы внутреннего блока A
      - - - -
      - - - -
      end block A;
B:    block
      - - - -      -- раздел описаний внутреннего блока B
      - - - -
      begin        -- выполнимые операторы внутреннего блока B
      - - - -
      - - - -
      end block B;
end XX;
```

В данном примере блоки A, B вложены в блок XX.

Обсудим теперь охранные выражения блоков. Рассмотрим архитектурное тело add1_e одноразрядного сумматора в виде охраняемого блока.

```
entity add1_e is
port (b1, b2, enable : in BIT;
      c1, s1 : out BIT);
end add1_e;
architecture struct_3 of add1_e is
begin
p0: block (enable = '1')
begin
s1<= guarded (b1 xor b2);
c1<= guarded (b1 and b2);
end block p0;
end struct_3;
```

Охранным выражением блока является выражение enable = 1. Если это выражение принимает значение true (истина), то охраняемые конструкции (назначения сигналов) выполняются, т. е. одноразрядный сумматор складывает числа, если же значение выражения является false (ложь), то охраняемые назначения сигналов не выполняются, т. е. сумматор не складывает числа b1, b2. Охрана назначения сигналов осуществляется указанием ключевого слова guarded.

В качестве другого примера охраняемого блока приведем пример описания D-триггера с асинхронным сбросом в виде блока с охранным выражением (clk = '1' or clr = '1').

```
entity dlatch is
port ( D, clk, clr : in bit;   Q : out bit);
end dlatch;
architecture functional of dlatch is
begin
      P: block (clk = '1' or clr = '1')
```

```
        begin
        Q <= guarded '0' when clr = '1' else
                        D when clk = '1' else
                    unaffected;
        end block P;
end functional;
```

В данном примере clk — вход синхронизации, clr — асинхронный сброс, D — вход данных, Q — выход триггера. Когда охранное выражение (clk = '1' **or** clr = '1') имеет значение ложь, то сигнал Q в левой части сохраняет свое прежнее значение. Легко видеть, что сигнал асинхронного сброса имеет приоритет по отношению к сигналу clk. Ключевое слово **unaffected** употребляется в операторе условного назначения сигнала для случая, когда требуется, чтобы назначаемый сигнал (в примере сигнал Q) не изменял своего значения.

УПРАЖНЕНИЯ

1. Перечислите последовательные операторы языка VHDL.
2. Какой порядок выполнения последовательных операторов?
3. Правильно ли утверждение: «Булево условие в цикле типа while проверяется в начале каждой итерации»?
4. Правильно ли утверждение: «Счетчик в цикле типа for есть переменная, которую нужно декларировать в начале процесса, в котором цикл употребляется»?
5. Какой будет дельта-задержка после выполнения операторов в случаях

a) X:= A + B + C + D;
b) Y:= A + B + C;
c) Z:= A + B;
d) W:= A?

6. Где в VHDL-коде может быть декларирована локальная переменная?

7. Где в VHDL-коде может быть декларирован сигнал?

8. Перечислите различия между локальными переменными и сигналами в языке VHDL .

9. Какие из последовательных операторов могут быть помечены? Являются ли метки обязательными?

10. Может ли процесс иметь список сигналов запуска и оператор **wait** внутри оператора процесса?

11. Может ли процесс иметь

– несколько списков сигналов запуска;

– несколько операторов **wait** внутри оператора процесса;

– несколько списков сигналов запуска и несколько операторов **wait** внутри оператора процесса?

12. Правильно ли, что знак оператора назначения сигнала может быть ориентирован как в левую сторону (<=), так и в правую сторону (=>) по желанию проектировщика, пользующегося языком VHDL?

13. Объясните, как Вы понимаете запись на языке VHDL

x <= y <= z

Является ли она корректной? Почему?

14. Какая часть VHDL-кода содержит последовательные операторы? Выберите правильный ответ:

a) процесс (перед ключевым словом begin);
b) архитектурное тело;
c) процесс (после ключевого слова begin);
d) пакет.

15. Пусть имеется фрагмент VHDL-кода.

```
signal a_bus : bit_vector( 3 downto 0);
signal z_bus : bit_vector( 3 downto 0);
signal a_bit, b_bit, c_bit, d_bit : bit;

1) BYTE <= (OTHERS => '1');
2) z_bus <= a_bit & b_bit;
3) a_bus <= ('1', b_bit, '0', d_bit);
4) a_bus (0 to 1) <= (OTHERS => '0');
```

Какие строки корректны? Выберите правильный ответ. Обоснуйте ответ:

a) только 2 и 4;
b) только 3 и 4;
c) только 1 и 2;
d) только 1 и 3.

16. Пусть имеется фрагмент VHDL-кода.
Type my_state **is** (RESET, IDLE, RW_CYCLE, INT_CYCLE);

signal STATE : my_state;
signal TWO_BIT : bit_vector (0 to 1);
1) STATE <= RESET;
2) STATE <= "00";
3) STATE <= TWO_BIT;

Какие строки корректны?

17. В какой части VHDL-кода можно употреблять операторы if, case, for ... loop? Выберите правильный ответ:

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

18. В данном фрагменте VHDL-кода осуществляется инициализация массива z_bus четырьмя способами:

Signal z_bus : bit_vector (3 **downto** 0);

a) z_bus <= "0000";
b) z_bus <= (1 => '0' , **others** => '0');
c) z_bus <= (**others** => '0');
d) z_bus <= ('0' ,'0','0', '0');

Укажите способ, который наиболее просто осуществляет инициализацию массива независимо от его длины.

19. Что такое транспортная и инерционная задержка сигнала? Какой тип задержки (транспортная, инерционная) принят по умолчанию в языке VHDL?

20. В каком случае модель инерционной задержки сигнала и модель транспортной задержки сигнала дают тот же результат?

21. Перечислите параллельные операторы языка VHDL.

22. Какая часть VHDL-кода содержит параллельные операторы назначения сигнала? Выберите правильный ответ:

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

23. Какой параллельный оператор может быть внутри параллельных операторов?

24. В следующем VHDL-коде [9] имеются четыре параллельных вызова процедуры:

```
entity PROCALL_EX is
end PROCALL_EX;
architecture RTL of PROCALL_EX is
  procedure ANDOR (
    signal A, B, C, D : in  bit_vector(1 downto 0);
    signal Y          : out bit_vector(1 downto 0)) is
  begin
    Y <= (A and B) or (C and D);
  end ANDOR;   signal DIN, DOUT : bit_vector(7 downto 0);
  signal X, Y, Z  : bit_vector(1 downto 0);
begin
call0 : ANDOR (A => DIN(7) & DIN(6), B => DIN(5 downto 4),
               C => DIN(3 downto 2), D => DIN(1 downto 0),
               Y => DOUT(1 downto 0));
call1 : ANDOR (A => DIN(7 downto 6), B => DIN(5 downto 4),
```

```
            C => DIN(3 downto 2), D => DIN(1 downto 0),
            Y => DOUT(3 downto 2));
call2 : ANDOR (A => DIN(7 downto 6) and DIN(5 downto 4),
            B => DIN(5 downto 4),
            C => DIN(3 downto 2), D => DIN(1 downto 0),
            Y => DOUT(5 downto 4));
call3 : ANDOR (A => X nand Y, B => Z,
            C => DIN(3 downto 2), D => DIN(1 downto 0),
            Y => DOUT(7 downto 6));
end RTL;
```

Является ли каждый вызов корректным или нет, почему?

25. Может ли локальная переменная употребляться как фактический параметр оператора параллельного вызова процедуры? Составьте соответствующий VHDL-код, проверьте его с помощью VHDL-анализатора.

26. Каким параллельным операторам требуются метки?

27. Для каких параллельных операторов метки необязательны?

28. Что такое компонент в языке VHDL? Чему соответствует компонент в логической схеме? Какие средства языка VHDL употребляются для соединения компонент?

29. Могут ли декларироваться компоненты внутри раздела декларации процесса?

30. Требуется ли декларировать сигналы внутри процесса?

31. Допустимо ли декларировать любые объекты внутри процесса?

32. Правильно ли, что все операторы внутри процесса выполняются один за другим?

33. Правильно ли, что все процессы внутри архитектурного тела выполняются один за другим?

34. Должен ли каждый процесс иметь имя (метку)? Рекомендуется ли именовать процессы? Если «да», то почему?

35. Правильно ли, что имя процесса специфицируется после ключевого слова process?

36. Правильно ли, что как сигналы, так и переменные могут употребляться для хранения временных данных внутри процесса?

37. Перепишите следующий VHDL-код [9], используя оператор выборочного назначения сигнала.

```
entity IFCASE is
  port (
    HEX  : in  bit_vector(3 downto 0);
    LED  : out bit_vector(6 downto 0));
end IFCASE;
architecture RTL of IFCASE is
begin
  p0 : process (HEX)
  begin
    case HEX is
      when "0000" => LED <= "1111110";
      when "0001" => LED <= "1100000";
      when "0010" => LED <= "1011011";
      when "0011" => LED <= "1110011";
      when "0100" => LED <= "1100101";
      when "0101" => LED <= "0110111";
      when "0110" => LED <= "0111111";
      when "0111" => LED <= "1100010";
      when "1000" => LED <= "1111111";
      when "1001" => LED <= "1110111";
      when "1010" => LED <= "0111001";
      when "1011" => LED <= "0111101";
      when "1100" => LED <= "0011001";
      when "1101" => LED <= "1111001";
      when "1110" => LED <= "1011111";
      when others => LED <= "0001111";
    end case;
  end process;
end RTL;
```

Проведите моделирование, сравните результаты.

38. Правильно ли, что оператор выборочного назначения сигнала и оператор условного назначения сигнала представляют то же действие, только записанное по-разному?

39. Правильно ли утверждение «Условное назначение сигнала может употребляться как в архитектурном теле (как параллельный оператор), так и в процессе (как последовательный оператор)»?

40. Для чего служит оператор **generate**? Структуру каких схем удобно описывать с его помощью?

41. Допустимо ли при конкретизации компонентов использовать оба направления (=>, <=) в зависимости от того, является порт входным либо выходным?

42. Разработайте логическую схему и, употребляя оператор **generate,** запишите VHDL-код структурного описания мультиплексора с двумя, тремя, четырьмя и *n* управляющими входами.

43. Разработайте функциональное описание схемы для перемножения двух матриц A, B размерностью 4×4, элементами которых являются целые числа. Используйте двумерные массивы для входных и выходных данных.

Примените операторы generic, generate, запишите VHDL-код для общего случая, когда матрицы имеют размерность N × N.

Разработайте структурные описания для случаев N = 2, N = 3, используя умножители и сумматоры и полагая, что элементами a_{ij}, b_{ij} матриц A, B являются целые числа, не большие числа 3. Примените битовое представление входных и выходных данных.

Проведите моделирование, убедитесь в корректности VHDL-кода.

Введите соответствующие типы данных и разработайте поведенческое VHDL-описание схемы для умножения матрицы A на вектор c.

Разработайте структурное описание, используя битовое представление входных данных для случая, когда элементы матрицы A и компоненты вектора c являются целыми числами, не большими 7, а матрица A имеет размерность N × N, вектор c имеет размерность N, N = 2, 3, 4. Используйте функции преобразования типов integer, bit_vector.

44. На рис. 2.6 изображена схема 4-разрядного умножителя. Введите имена сигналам, соединяющим элементы схемы: одноразрядные полусумматоры add1, одноразрядные сумматоры add2, двухвходовые конъюнкторы.

Опишите схему, используя операторы создания экземпляров компонентов. Составьте более компактное описание, используя другие операторы языка VHDL.

Проведите моделирование, сравните результаты. Введите задержки элементам: add1 — 10ns, add2 — 15 ns, элементам И — 5 ns. Проведите моделирование. Определите задержку схемы.

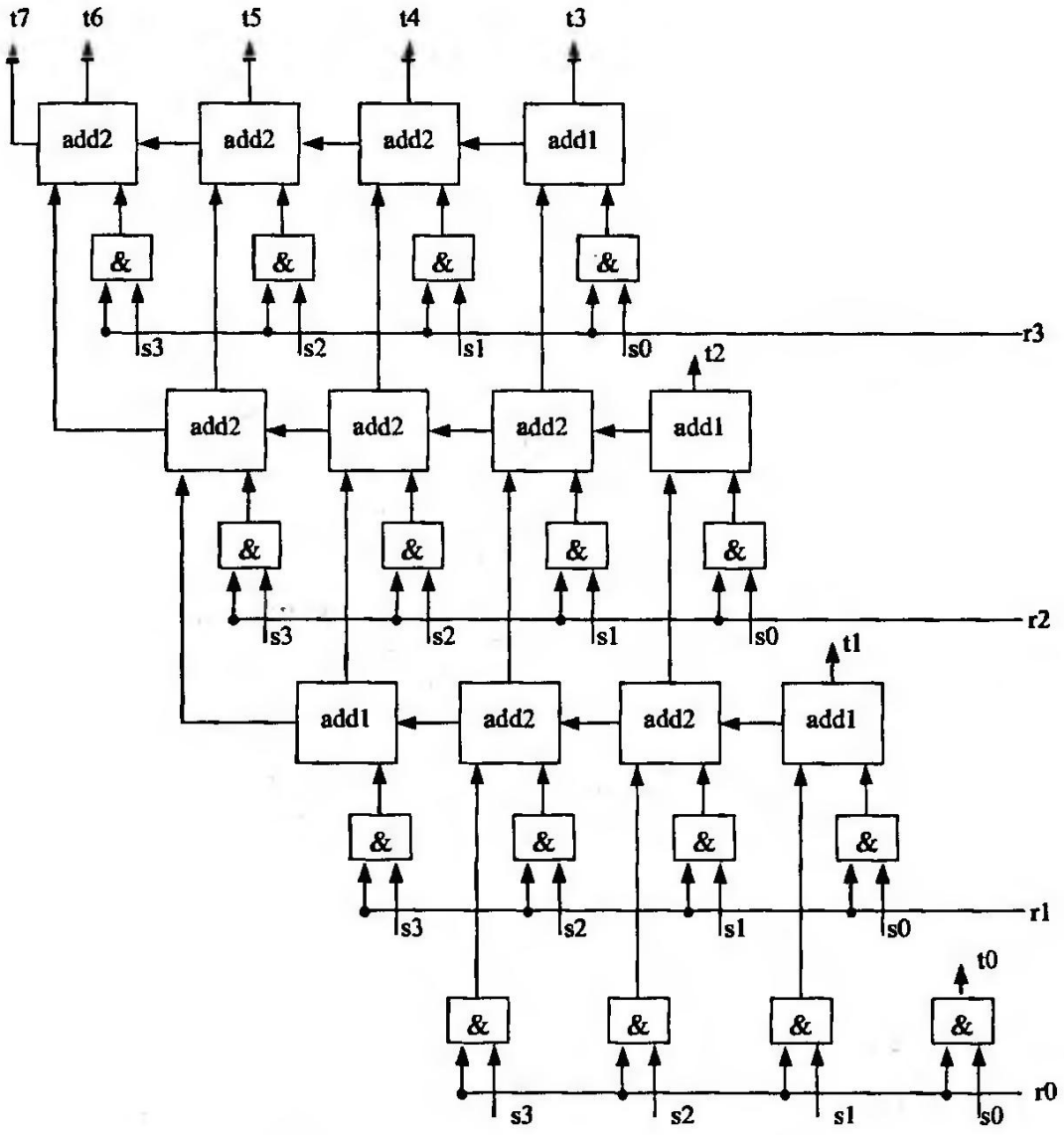

Рис. 2.6. Четырехразрядный умножитель

45. На рис. 2.7 дана схема D-триггера. Введите задержки элементам и составьте структурное описание. Модифицируйте функциональное описание, введя задержку в оператор назначения сигнала, проведите моделирование в обоих случаях. Сравните результаты.

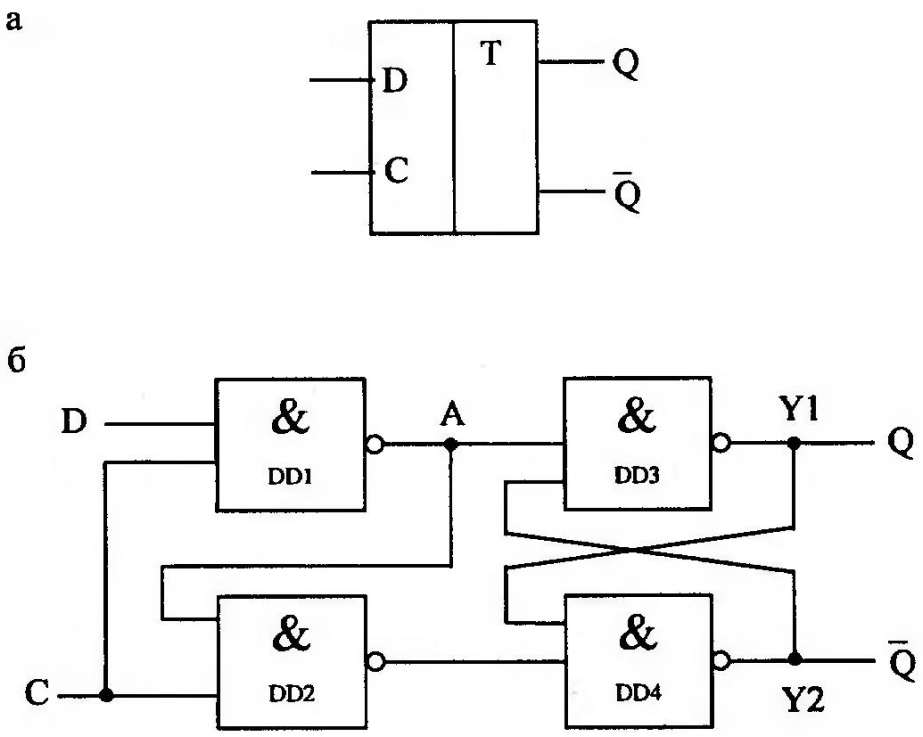

Рис. 2.7. D-триггер (защелка): а — условное обозначение; б — схема D-триггера на элементах И-НЕ

46. Опишите функционирование RS-триггера с использованием конструкции «охраняемый блок».