Глава 4

Примеры проектирования на VHDL

4.1. Стили описания поведения

Рассмотрим различные стили описания одного и того же поведения на примере дешифратора DC(2), который имеет два входных полюса и четыре выходных (рис. 4.1) .

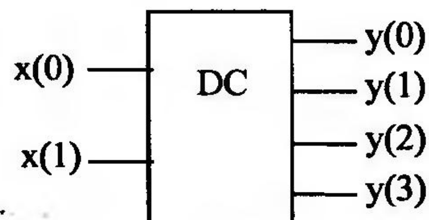

Рис. 4.1. Дешифратор

На рис. 4.1 слева и справа указаны имена входных и выходных полюсов, употребляемых в VHDL-описании. Функции дешифратора имеют следующий вид:

$$y0 = \overline{x0} \wedge \overline{x1};$$
$$y1 = \overline{x0} \wedge x1;$$
$$y2 = x0 \wedge \overline{x1};$$
$$y3 = x0 \wedge x1;$$

Интерфейс дешифратора

```
entity Decoder is
port      (X: in Bit_vector (0 to 1);
```

```
        Y: out Bit_vector (0 to 3));
end Decoder;
```

Представим различные *стили* описания функционирования дешифратора: «чисто» структурное описание, описание в виде потока данных, процедурное описание. Заметим, что при спецификации цифровых систем допустим смешанный стиль, что весьма удобно при проектировании.

Структурное описание архитектурного тела в виде схемы в базисе инверторов (Inverter) и двухвходовых элементов И (AND_Gate) имеет вид

```
architecture structure of Decoder is
signal S: bit_vector (0 to 1);
component AND_Gate port      (A, B: in Bit;  D: out Bit);
end component;
component Inverter port (A: in Bit;  B: out Bit);
end component;
begin
Inv1: Inverter port map (A => x(0), B => s(0));
Inv2: Inverter port map (A => x(1), B => s(1));
A1:AND_Gate port map (A => s(0), B => s(1), D => y(0));
A2:AND_Gate port map (A => s(0), B => x(1), D => y(1));
A3:AND_Gate port map (A => x(0), B => s(1), D => y(2));
A4:AND_Gate port map (A => x(0), B => x(1), D => y(3));
end structure;
```

Описание поведения дешифратора в виде ***потока данных*** (data flow).

```
architecture Data_flow of Decoder is
begin
y(0) <= not x(0) and not x(1);
y(1) <= not x(0) and x(1);
y(2) <= x(0) and not x(1);
y(3) <= x(0) and x(1);
end Data_flow;
```

Процедурное описание дешифратора выглядит следующим образом:

```
architecture Procedural of Decoder is
signal s: bit_vector (0 to 3);
begin
process (x)
begin
      case x is
            when "00" => s <= "1000";
            when "01" => s <= "0100";
            when "10" => s <= "0010";
            when "11" => s <= "0001";
      end case; end process;
      y <= s;
end Procedural;
```

Смешанное описание, использующее элементы структурного описания и элементы описания «поток данных», приведено ниже.

```
architecture Mixed of Decoder is
component Inverter port (A: in Bit; B: out Bit);
end component;
signal S: bit_vector (0 to 1);
begin
Inv1: Inverter port map (A => x(0), B => s(0));
Inv2: Inverter port map (A => x(1), B => s(1));
p: process (s, x)
      begin
y(0) <= s(0) and s(1);
y(1) <= s(0) and x(1);
y(2) <= x(0) and s(1);
y(3) <= x(0) and x(1);
end process;
end Mixed;
```

В данном примере структурное описание и описание типа «поток данных» весьма схожи, так как происходит замена логических элементов (компонентов) логическими выражениями. Однако для более сложных схем такая замена может быть совершенно не очевидной.

4.2. Формы описания сигналов

Битовое представление сигналов может быть громоздким. На начальных этапах проектирования могут быть использованы другие типы данных [1].

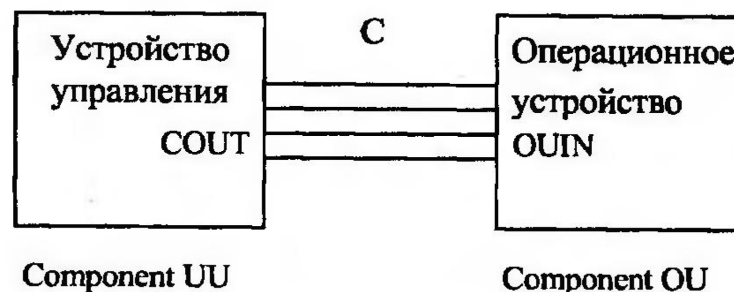

Рис. 4.2. Сигнал С может быть описан как bit_vector и как перечислимый тип

Пусть устройство управления (UU) посылает четырехбитовый код в операционное устройство (OU). Четырехбитовый код определяет операцию, подлежащую выполнению. На рис. 4.2 выходной порт устройства управления имеет имя COUT, входной порт операционного устройства — имя OUIN.

Можно описать сигнал С на битовом уровне

```
signal C: BIT_VECTOR(0 to 3);
```

или же как перечислимый тип

```
type CONTROL is (AND, OR, XOR, ADD, SUB, ..., TCOMP);
signal C: CONTROL;
```

Таким образом, если использовать перечислимый тип, то управляющую информацию можно представлять в мнемоническом

виде, т. е. именовать команды UU. При этом в архитектурном теле устройства управления можно было бы использовать операторы назначения сигнала, такие как

```
COUT <= ADD;
```

а в архитектурном теле операционного устройства

```
if (OUIN = ADD) then ...
```

При таком подходе разработчик модели освобождается от необходимости беспокоиться о низкоуровневых деталях [1].

Другой вариант описания сигнала С выглядит следующим образом:

```
subtype CONTROL is INTEGER range 0 to 15;
signal C: CONTROL;
```

Подтип CONTROL ограничивает диапазон значений, которые могут быть присвоены сигналу C, тем самым обеспечивается некоторый контроль ошибок.

В некоторых случаях может понадобиться как битовый уровень, так и более высокий уровень — типа INTEGER. Для преобразования типов нужны функции. Ниже описаны две функции.

Функция BIN4_TO_INT преобразует битовые векторы в целые числа путем суммирования соответствующих степеней двойки. Функция INT_TO_BIN4 выполняет обратную операцию.

Функция INT_TO_BIN4 приведена ниже в пакете vv_vls. Предоставляем читателю самостоятельно написать VHDL-код функции BIN4_TO_INT, для образца можно воспользоваться VHDL-кодом функции BIN2_TO_INT, находящемся в теле пакета vv_vls.

С использованием функций BIN2_TO_INT, INT_TO_BIN4 легко описывается (на алгоритмическом уровне) поведение схем mult_2, adder_2, входящих в схему VLSI_1 (см. гл. 1). Определим функции BIN2_TO_INT, INT_TO_BIN4 в пакете vv_vls.

```
package vv_vls is
function bin2_to_int (signal w2, w1: bit)
return integer;
function int_to_bin4
(signal INPUT : integer)
return bit_vector;
```

```
type fsm_in_type is (z1, z2);
type fsm_out_type is (w1, w2, w3, w4, w5);
end vv_vls;

package body vv_vls is
function bin2_to_int      -- функция преобразования битового
(signal w2, w1: bit)      -- вектора в число, w2 — старший разряд
return integer is
variable sum : integer := 0;
begin
if w1 = '1' then   sum := 1;
else   sum := 0;
end if;
if w2 = '1' then
sum := sum + 2;
else
sum := sum;
end if;
return sum;
end bin2_to_int;

function int_to_bin4              -- функция преобразования числа
(signal INPUT : integer)          -- в битовый вектор
return bit_vector is
variable fout: bit_vector(0 to 3);
variable temp_a: integer:= 0;
variable temp_b: integer:= 0;
begin
temp_a:= INPUT;
for i in 3 downto 0 loop
temp_b:= temp_a/(2 ** i);
temp_a:= temp_a rem (2 ** i);
if (temp_b = 1) then
fout(i):= '1';   else
fout(i):= '0';   end if;
end loop;
return fout;
```

```
end int_to_bin4;
end vv_vls;

library work; use work.vv_vls.all;
entity mult_2 is           -- функциональное описание умножителя
port (s1, s0, r1, r0 : in BIT;
      t3, t2, t1, t0 : out BIT);
end mult_2;
architecture functional of mult_2 is
signal ss, rr : integer range 0 to 3;
signal tt : integer range 0 to 9;
signal tt_sig : bit_vector (0 to 3);
begin
ss <= bin2_to_int(s1, s0);
rr <= bin2_to_int(r1, r0);
tt <= ss * rr;                    -- функция умножителя
tt_sig <= int_to_bin4(tt);
t0 <= tt_sig(0);                  -- формирование выходных сигналов
t1 <= tt_sig(1);
t2 <= tt_sig(2);
t3 <= tt_sig(3);
end functional;

library work; use work.vv_vls.all;
entity adder_2 is          -- функциональное описание сумматора
port (a1, b1, a2, b2 : in BIT;
      c2, s2, s1 : out BIT);
end adder_2;
architecture functional of adder_2 is
signal aa, bb : integer range 0 to 3;
signal cc : integer range 0 to 7;
signal cc_sig : bit_vector (0 to 3);
begin
aa <= bin2_to_int (a1, b1);
bb <= bin2_to_int (a2, b2);
cc <= aa + bb;                    -- функция сумматора
cc_sig <= int_to_bin4(cc);
```

```
s1 <= cc_sig(0);                -- формирование выходных сигналов
s2 <= cc_sig(1);
c2 <= cc_sig(2);
end functional;
```

Предлагаем читателю провести моделирование схемы VLSI_1 (разд. 1.1), используя архитектурные тела functional (алгоритмические описания) для объектов проекта mult_2, adder_2 вместо архитектурных тел structure, представленных ранее в разд. 1.1. Чтобы заменить в VHDL-коде архитектурного тела functional схемы VLSI_1 алгоритмическое описание структурным, необходимо сначала перейти к битовому уровню представления входных сигналов a, b (использовать функцию преобразования типа INTEGER в тип BIT_VECTOR), а затем от битового представления выходных сигналов D перейти к типу INTEGER. Предоставляем читателю сделать это самостоятельно.

4.3. Описание автоматов

Схема с обратной связью

Рассмотрим схему, изображенную на рис. 4.3. В нее входит элемент памяти (D-триггер) и комбинационный элемент И. Данная схема описывается в виде двух процессов, причем для описания функционирования D-триггера вводится функция rising_edge.

```
architecture circ_feedback  of some_entity is
signal b: bit;
function rising_edge (signal s : bit) return boolean is
   begin
      return s = '1' and s'event;   end;
begin
      process (clk, reset)
 begin
 if reset = '1' then
 c <= '0';
elsif rising_edge(clk) then
c <= b;
end if;
```

```
 end process;
process (a, c)                -- комбинационный процесс
begin b <= a and c;
 end process;
end circ_feedback ;
```

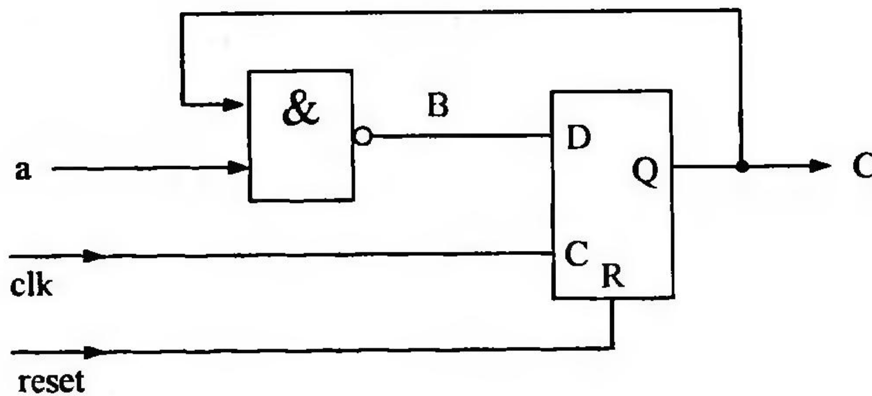

Рис. 4.3. Схема с обратной связью
(architecture circ_feedback)

Конечный автомат

Широкое распространение в практике проектирования дискретных устройств получила модель конечного (абстрактного) автомата.

Конечный автомат K определяется как набор

$$K = (A, Z, W, \delta, \lambda, a_1),$$

где $A = \{a_1, ..., a_Q\}$ — множество (алфавит) состояний (имеются в виду внутренние состояния автомата);

$Z = \{z_1, ..., z_N\}$ — множество входных сигналов (входной алфавит);

$W = \{w_1, ..., w_M\}$ — множество выходных сигналов (выходной алфавит);

δ — функция переходов, определяющая состояние автомата в момент времени t + 1 в зависимости от состояния автомата и входного сигнала в момент времени t, иначе говоря,

$$a_s = \delta (a_m , z), \quad (4.1)$$

где a_m — состояние автомата в момент времени t; z — входной сигнал в момент времени t; a_s — состояние автомата в момент времени t + 1;

λ — функция выходов. Если функция λ по состоянию автомата a_q и входному сигналу z_n определяет значение выходного сигнала w_m

$$w_m = \lambda (a_q , z_n), \quad (4.2)$$

то конечный автомат называется *автоматом Мили* , если же

$$w_m = \lambda (a_q), \quad (4.3)$$

то конечный автомат называется *автоматом Мура*;

a_1 — начальное состояние автомата в момент времени t = 0, $a_1 \in A$.

Функционирование автомата происходит следующим образом: в дискретные моменты времени t = 0, 1, 2, 3, ... на вход устройства поступает входной сигнал — один из элементов множества Z, а на выходе появляется выходной сигнал — один из элементов множества W, в этот же момент времени t автомат из состояния a_m переходит в состояние a_s, в котором он будет находиться в момент времени t + 1.

Разработаны различные формы задания конечных автоматов. Например, в табл. 4.1 задан конечный автомат Мили с четырьмя внутренними состояниями [2].

Таблица 4.1

Входные сигналы	Состояния			
	a_1	a_2	a_3	a_4
z_1	a_2/w_1	a_2/w_1	a_1/w_2	a_1/w_4
z_2	a_4/w_5	a_3/w_3	a_4/w_4	a_3/w_5

Алфавит состояний $A = \{a_1, a_2, a_3, a_4\}$, $q = 1, ..., 4$. Входной алфавит Z образуют сигналы z_1, z_2, т. е. $Z = \{z_1, z_2\}$, $n = 1,2$. Выходной алфавит W образуют сигналы $w_1, ..., w_5$, т. е. $W = \{w_1, w_2, w_3, w_4, w_5\}$, $m = 1, ..., 5$. На пересечении строки z_n и столбца a_q в таблице находится состояние a_s, в которое должен перейти автомат из состояния a_q под воздействием сигнала z_n. После косой черты в этой же графе таблицы указывается выходной сигнал, выдаваемый автоматом в состоянии a_q при поступлении на его вход сигнала z_n. Иначе говоря, табл. 4.1 является таблицей задания функций δ, λ и называется *совмещенной таблицей переходов и выходов.*

Автомат может быть задан ориентированным графом, в котором вершинам соответствуют абстрактные внутренние состояния автомата, а дуги соответствуют переходам между состояниями. Такой граф носит название *графа автомата*. Для автомата, заданного табл. 4.1, граф автомата имеет вид (рис.4.4).

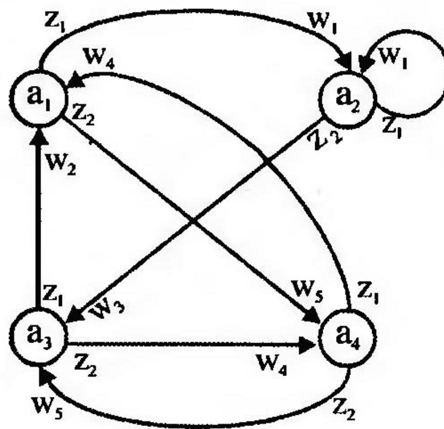

Рис. 4.4. Граф конечного автомата

Легко видеть, что совмещенная таблица переходов и выходов автомата и граф автомата являются эквивалентными формами задания поведения конечного автомата Мили.

Следующий VHDL-код реализует поведение конечного автомата, заданного в табл. 4.1. Поведение автомата представляется в виде совокупности двух взаимодействующих процессов (рис. 4.5).

Процесс NS определяет выходной сигнал w и внутренний сигнал NEXT_state, который является входным для процесса REG. Заметим, однако, что в списке чувствительности процесса REG имеется только сигнал clk, который соответствует моментам времени срабатывания автомата.

Для входных сигналов автомата в пакете vv_vls определяется перечислимый тип

type fsm_in_type **is** (z1, z2);

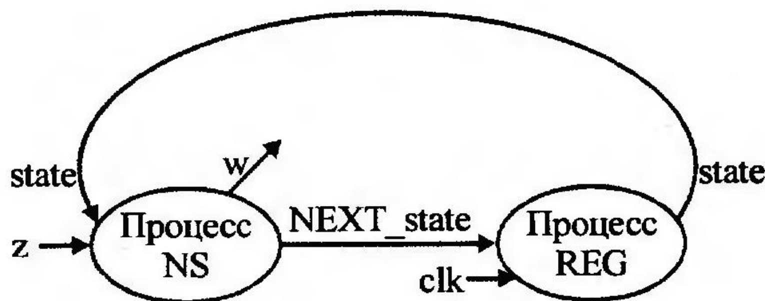

Рис. 4.5. Описание функционирования конечного автомата в виде совокупности двух процессов (entity FSM_A)

для выходных сигналов в том же пакете определяется перечислимый тип

type fsm_out_type **is** (w1, w2, w3, w4, w5);

Для задания состояний автомата в архитектурном теле определяется перечислимый тип t_state. Предполагается, что начальное состояние автомата равно a1. Начальное состояние явно не указывается, так как при инициализации будет взят элемент нижней границы перечислимого типа — это и будет элемент a1.

```
Library work; use work.vv_vls.all;
entity FSM_A is
  port (z : in fsm_in_type;
    clk : in bit;
```

```
        w : out fsm_out_type);
end FSM_A;

architecture rtl_a of fsm_a is
type T_state is (a1, a2, a3, a4);
signal NEXT_state : T_state;
signal state : T_state;

begin
NS : process (state, z)
begin
case state is
  when a1 =>
  if (z = z1) then NEXT_state <= a2; w <= w1;
  elsif (z = z2) then NEXT_state <= a4; w <= w5;
   end if;
  when a2 =>
  if (z = z1) then NEXT_state <= a2; w <= w1;
  elsif (z = z2) then NEXT_state <= a3; w <= w3;
   end if;
  when a3 =>
  if (z = z1) then NEXT_state <= a1; w <= w2;
  elsif (z = z2) then NEXT_state <= a4; w <= w4;
   end if;
  when a4 =>
  if (z = z1) then NEXT_state <= a1; w <= w4;
  elsif (z = z2) then NEXT_state <= a3; w <= w5;
  end if;
end case;          end process;
REG : process(clk)
begin
if clk = '1' then
state <= NEXT_state;
end if;
end process;
end rtl_a;
```

Сделав следующие назначения сигналов z, clk

```
z <= z1,  z2 after 50 ns,
          z1 after 100 ns,  z2 after 150 ns,
          z1 after 200 ns,  z2 after 250 ns;

clk <= '1',  '0' after 25 ns,   '1' after 50 ns,
             '0' after 75 ns,   '1' after 100 ns,
             '0' after 125 ns,  '1' after 150 ns,
             '0' after 175 ns,  '1' after 200 ns,
             '0' after 225 ns,  '1' after 250 ns;
```

«снаружи» объекта FSM_A, получим соответствующую временную диаграмму (рис. 4.6) функционирования автомата.

Многократное изменение синхросигнала clk можно более компактно описать в виде процесса generator. Переменная number задает число тактов. В данном случае число тактов равно 5.

```
generator : process
variable number : integer := 5;
begin
for i in 1 to number loop
clk <= transport '1'; wait for 25 ns;
clk <= transport '0'; wait for 25 ns;
end loop;
end process;
```

При схемной реализации автомата обычно поступают следующим образом. Выделяют комбинационную часть автомата и память. Как же детализировать описание поведения автомата на случай элементов памяти того или иного типа? В случае нашего примера будем реализовывать элементы памяти на D-триггерах. Заметим, что в архитектурном теле rtl_a для entity FSM_A не указывается, как будут функционировать элементы памяти. Так как поведение D-триггера нам известно, можем использовать соответствующий VHDL-код для описания процесса REG. Описание поведения конечного автомата в виде совокупности двух взаимодействующих процессов — процесса NS и детализированного процесса REG — показано на рис. 4.7.

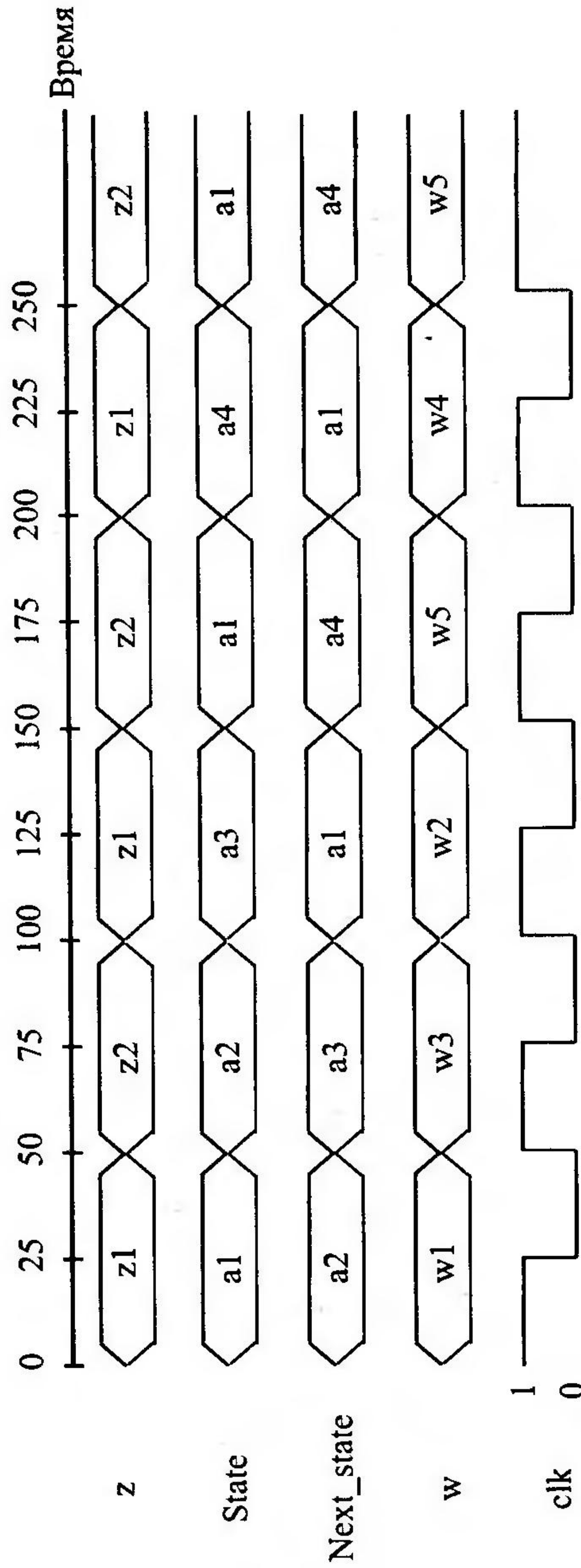

Рис. 4.6. Временная диаграмма функционирования конечного автомата

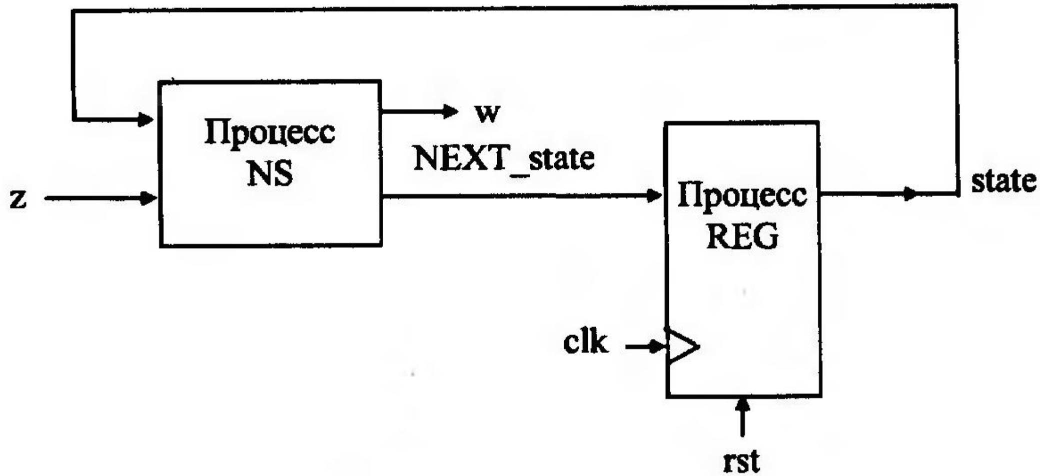

Рис. 4.7. Детализированное описание функционирования конечного автомата (entity FSM)

```
Library work;  use work.vv_vls.all;
entity FSM is
  port (z : in fsm_in_type;
      clk, rst : in bit;
      w : out fsm_out_type);
end FSM;
architecture rtl of fsm is
type T_state is (a1, a2, a3, a4);
signal NEXT_state, state : T_state;
begin
NS : process (state, z)      -- комбинационный процесс
. . .
-- описание процесса NS дано в архитектурном теле
-- rtl_a для entity FSM_A
end process NS;
REG: process (clk, rst)                -- процесс REG для D-триггера
begin
if (rst = '1') then state <= a1;
elsif clk'event and clk = '1' then
```

```
state <= NEXT_state;
end if;
end process REG;
end rtl;
```

Дальнейшая детализация поведения автомата связана с кодированием значений перечислимых типов булевыми векторами, описанием функций комбинационной части схемы и формированием регистра триггеров. В качестве упражнения можно составить VHDL-код для структурной схемы автомата, для этого можно использовать приведенную в [2] структуру: булевы функции для комбинационной части и регистр D-триггеров.

Микропрограммный автомат

Микропрограммный автомат является одной из форм задания конечного автомата для случая, когда абстрактные входные и выходные сигналы автомата закодированы булевыми переменными. Состояния микропрограммного автомата являются абстрактными (не закодированными). Микропрограммный автомат может задаваться графически — в виде граф-схемы алгоритма (ГСА). Одна и та же ГСА может реализовываться различными структурными схемами — автоматом Мили (Mealy) либо автоматом Мура (Moore). Получение различных структурных схем связано с различными методиками разметки ГСА — так называется процесс ввода внутренних состояний автомата. Далее будут приведены две реализации одной и той же ГСА (рис. 4.8) в виде автомата Мили и автомата Мура.

Пример ГСА взят из книги [2], в которой подробно рассматриваются методики кодирования состояний и перехода от ГСА либо к графу автомата (графу переходов между состояниями), либо к табличной форме задания размеченной ГСА.

Автомат Мили

На рис. 4.8 показаны внутренние состояния a_1, a_2, ..., a_6, соответствующие модели автомата Мили. Граф автомата Мили задан на рис. 4.9.

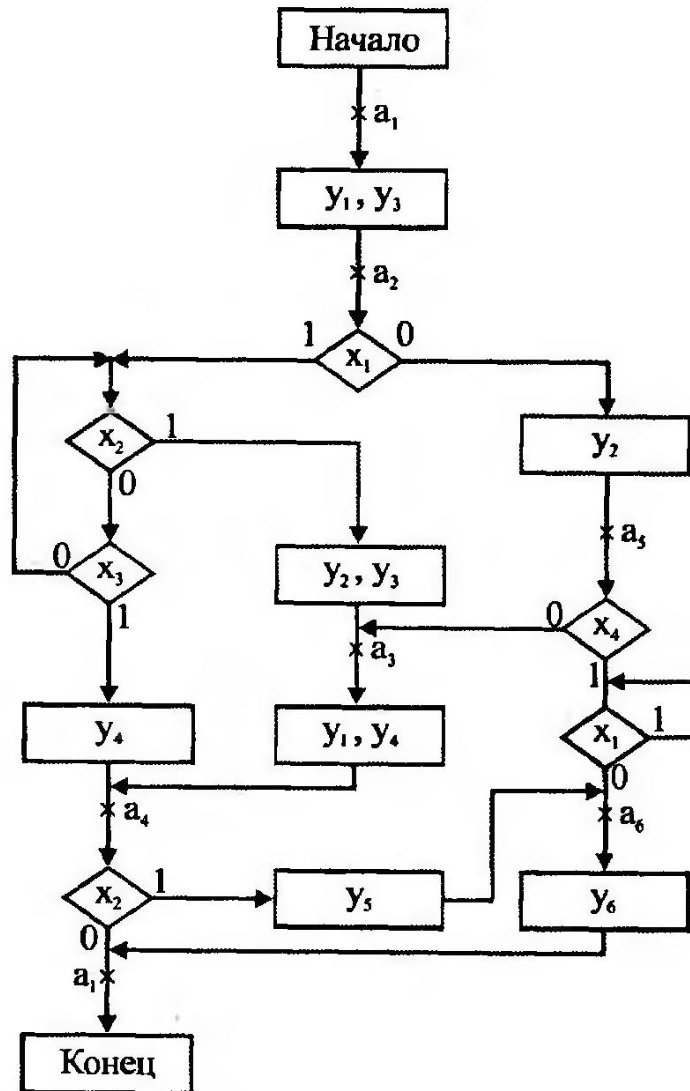

Рис. 4.8. Граф-схема алгоритма (разметка состояний для автомата Мили)

Таблица 4.2 представляет собой табличную (эквивалентную графовой) форму задания автомата Мили.

Таблица 4.2

Таблица переходов автомата Мили

Состояние a_m	Состояние a_s	Условия перехода	Выходные сигналы
a_1	a_2	1	y_1y_3
a_2	a_2	$x_1\bar{x}_2\bar{x}_3$	—
	a_3	x_1x_2	y_2y_3
	a_4	$x_1\bar{x}_2x_3$	y_4
	a_5	$\bar{x}_1$	y_2
a_3	a_4	1	y_1y_4
a_4	a_1	$\bar{x}_2$	—
	a_6	x_2	y_5
a_5	a_1	$\bar{x}_1x_4$	y_6
	a_4	$\bar{x}_4$	y_1y_4
	a_5	x_1x_4	—
a_6	a_1	1	y_6

В первом столбце табл. 4.2 показано состояние a_m микропрограммного автомата, из которого совершается переход, во втором столбце — состояние a_s, куда совершается переход. Переходу соответствует строка табл. 4.2.

VHDL-модель автомата Мили, построенная по эквивалентным формам задания (ГСА (рис.4.8), граф автомата (рис.4.9), табл. 4.2) приводится ниже.

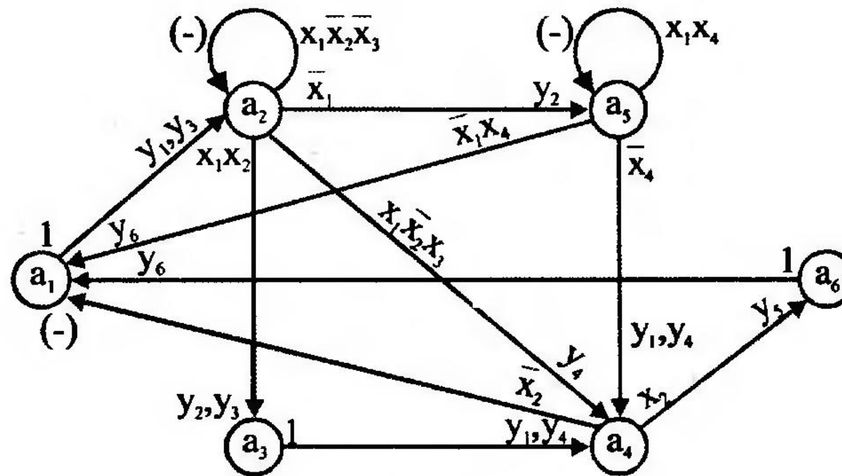

Рис. 4.9. Граф автомата Мили

```
entity Mealy is
port(x:in bit_vector (4 downto 1); clk, rst :in bit;
         y:out bit_vector (6 downto 1));
end Mealy;
architecture rtl of Mealy is
type T_state is (a1, a2, a3, a4, a5, a6);
signal NEXT_state, state:T_state;
begin
NS : process (state, x)
begin
case state is
when a1 =>
      NEXT_state <= a2; y <= "000101";
                          -- код y = (y6, y5, y4, y3, y2, y1)
                          -- код x = (x4, x3, x2, x1)
when a2 =>
      if ((x(1) and not x(2) and not x(3)) = '1')
               then NEXT_state <= a2; y <= "000000";
      elsif (not x(1) = '1')
               then NEXT_state <= a5; y <= "000010";
      elsif ((x(1) and not x(2) and x(3)) = '1')
           then NEXT_state <= a4; y <= "001000";
```

```
        elsif ((x(1) and x(2)) = '1')
                        then NEXT_state <= a3; y <= "000110";
          end if;
when a3 => NEXT_state <= a4; y <= "001001";
when a4 => if (x(2) = '1')
          then NEXT_state <= a6 ; y <= "010000";
               elsif (not x(2)  = '1')
          then NEXT_state <= a1 ; y <= "000000";
          end if;
when a5 =>
          if ((x(1) and x(4)) = '1')
          then NEXT_state <= a5; y <= "000000";
               elsif (not x(4) = '1')
          then NEXT_state <= a4; y <= "001001";
               elsif ((not x(1) and x(4)) = '1')
          then NEXT_state <= a1; y <= "100000";
          end if;
when a6 => NEXT_state <= a1; y <= "100000";
end case;
end process NS;
state <= a1 when rst = '1' else
NEXT_state when clk'event and clk = '1' else state;
end rtl;
```

В формальной модели микропрограммного автомата (граф-схеме алгоритма) сигналы clk, rst отсутствуют. Введение сигнала clk в текст VHDL-программы необходимо для управления переключением элементами памяти по переднему фронту сигнала clk. Можно обойтись без оператора процесса, а воспользоваться оператором назначения сигнала для переключения состояний. В тексте VHDL-кода присутствует также сигнал rst, управляющий установкой автомата в начальное состояние a1.

Автомат Мура.

На рис. 4.10 показаны внутренние состояния a_1, a_2, ..., a_9, соответствующие модели автомата Мура.

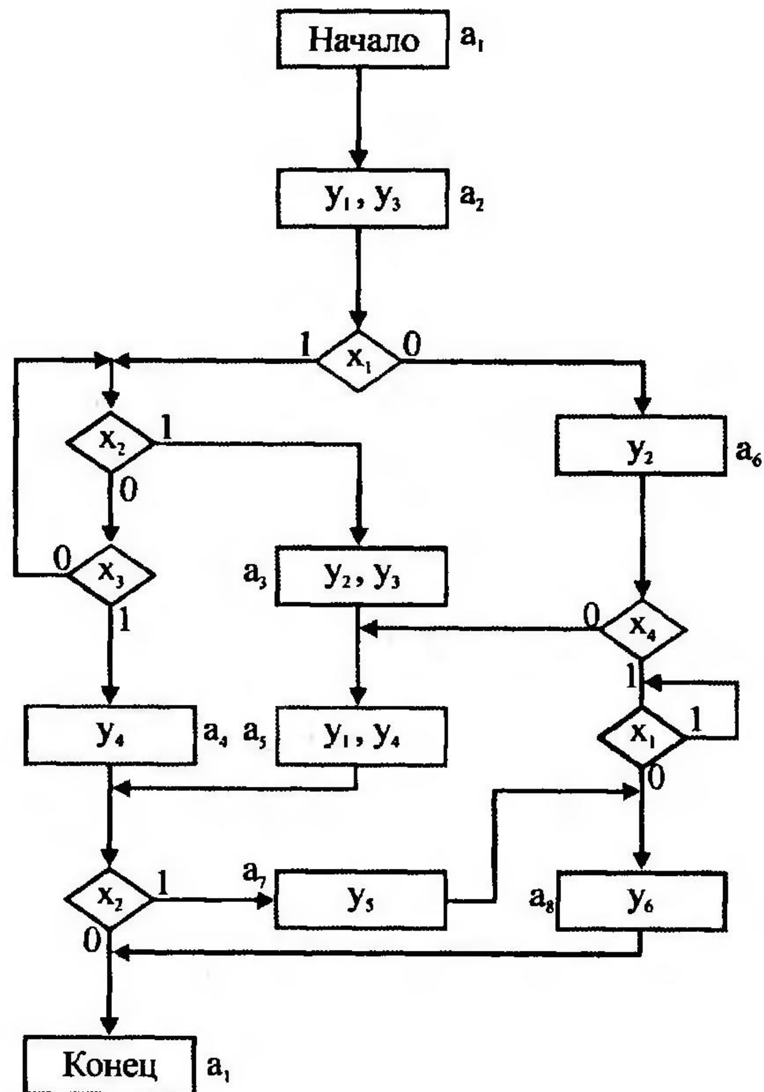

Рис. 4.10. Граф-схема алгоритма (разметка состояний для автомата Мура)

Граф автомата Мура задан на рис. 4.11, табл. 4.3 представляет собой эквивалентную размеченной ГСА графовую форму задания автомата Мура. В первом столбце табл. 4.3 показано состояние микропрограммного автомата, из которого совершается переход, во втором столбце — состояние, куда совершается переход. Так как для автомата Мура выходные сигналы определяются только состоянием, то для каждого состояния во втором столбце указываются те выходные сигналы, единичные значения которых автомат выдает, находясь в данном состоянии.

Таблица 4.3

Таблица переходов автомата Мура

Состояние a_m	Состояние a_s, выходные сигналы	Условие перехода
a_1	a_2	1
a_2	a_2, y_1y_3	$x_1\bar{x}_2\bar{x}_3$
a_2	a_3, y_2y_3 a_4, y_4 a_6, y_2	x_1x_2 $x_1\bar{x}_2x_3$ $\bar{x}_1$
a_3	a_5, y_1y_4	1
a_4	a_1 a_7, y_5	$\bar{x}_2$ x_2
a_5	a_1 a_7, y_5	$\bar{x}_2$ x_2
a_6	a_5, y_1y_4 a_6, y_2 a_8, y_6	$\bar{x}_4$ x_1x_4 $\bar{x}_1x_4$
a_7	a_8, y_6	1
a_8	a_1	1

VHDL-модель автомата Мура, построенная по эквивалентным формам задания (ГСА (рис. 4.10), граф автомата (рис. 4.11), табл. 4.3) приводится ниже. Сигнал clk управляет переключениями элементов памяти, по единичному значению сигнала rst автомат переходит в начальное состояние a1.

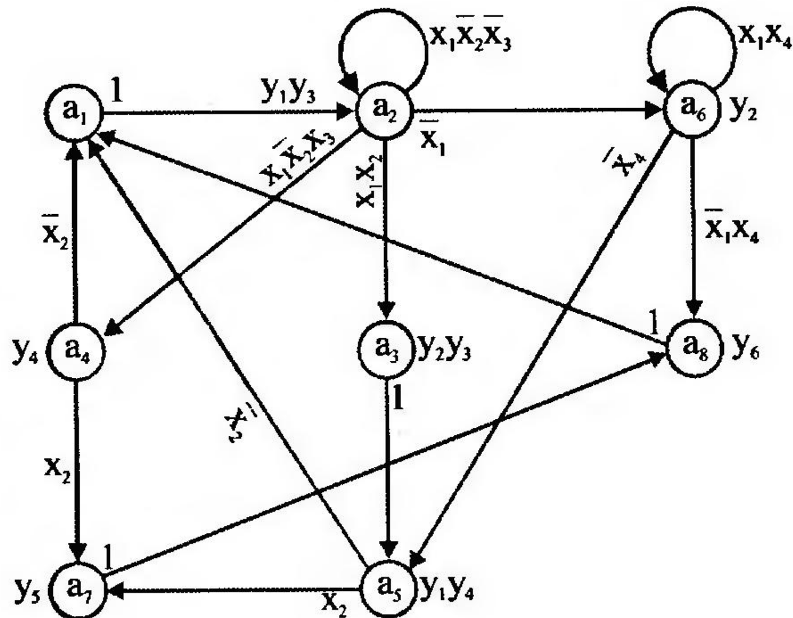

Рис. 4.11. Граф автомата Мура

```
entity Moore is
port(x:in bit_vector (4 downto 1);
        clk, rst :in bit;
        y:out bit_vector (6 downto 1));
end Moore;
architecture rtl of Moore is
type T_state is (a1, a2, a3, a4, a5, a6, a7, a8);
signal NEXT_state, state:T_state;
begin
NS : process (state, x)
begin
case state is
when a1 => NEXT_state <= a2;
when a2 =>
        if ((x(1) and not x(2) and not x(3)) = '1')
        then NEXT_state <= a2;
                elsif ((x(1) and x(2)) = '1')
```

```
        then NEXT_state <= a3;
                elsif ((x(1) and not x(2) and x(3)) = '1')
        then NEXT_state <= a4;
                elsif (not x(1) = '1')
        then NEXT_state <= a6;
        end if;
when a3 =>NEXT_state <= a5;
when a4 =>
        if (not x(2) = '1') then NEXT_state <= a1;
                elsif (x(2) = '1')
        then NEXT_state <= a7;
        end if;
when a5 =>
        if (not x(2) = '1') then NEXT_state <= a1;
                elsif (x(2) = '1') then NEXT_state <= a7;
        end if;
when a6 =>
        if (not x(4) = '1') then NEXT_state <= a5;
                elsif ((x(1) and x(4)) = '1')
        then NEXT_state <= a6;
                elsif ((not x(1) and x(4)) = '1')
        then NEXT_state <= a8;
        end if;
when a7 =>   NEXT_state <= a8;
when a8 =>   NEXT_state <= a1;
end case;
end process NS;

y <=    "000000" when state = a1 else
        "000101" when state = a2 else
        "000110" when state = a3 else
        "001000" when state = a4 else
        "001001" when state = a5 else
        "000010" when state = a6 else
        "010000" when state = a7 else
        "100000";
```

```
state <= a1 when rst = '1' else
 NEXT_state when clk'event and clk = '1' else state;
end rtl;
```

Обращаем внимание на то, что в автомате Мура выходные сигналы полностью определяются внутренними состояниями автомата, поэтому их можно «собрать» в один оператор параллельного назначения сигнала.

4.4. Отладка VHDL-описаний

Отладка VHDL-кодов происходит путем подачи входных сигналов в объект проекта, после чего происходит сравнение полученных реакций с ожидаемыми. Этот процесс, по существу, аналогичен отладке программ. С точки зрения «отладки» аппаратуры этот процесс напоминает тестирование схемы на предмет установления правильности ее функционирования.

Пример тестирования VHDL-кода схемы VLSI_1.

```
entity test_vlsi_1 is
end test_vlsi_1;

architecture test of test_vlsi_1 is
component vlsi_1
port (a2, a1, b2, b1, x : in BIT;
    d4, d3, d2, d1   : out BIT);
end component;
 signal a2, a1, b2, b1, x, d4, d3, d2, d1 : BIT;
 begin
p:  vlsi_1
port map (a2 => a2, a1 => a1, b2 => b2, b1 => b1, x => x,
        d4 => d4, d3 => d3, d2 => d2, d1 => d1);
b2 <= '1'; b1 <= '1'; a2 <= '1'; a1 <= '0', '1' after 50 ns;
 x <= '0', '1' after 30 ns, '0' after 70 ns ;
end test;
```

Временная диаграмма показана на рис. 4.12. В табл. 4.4 дано функционирование схемы во времени.

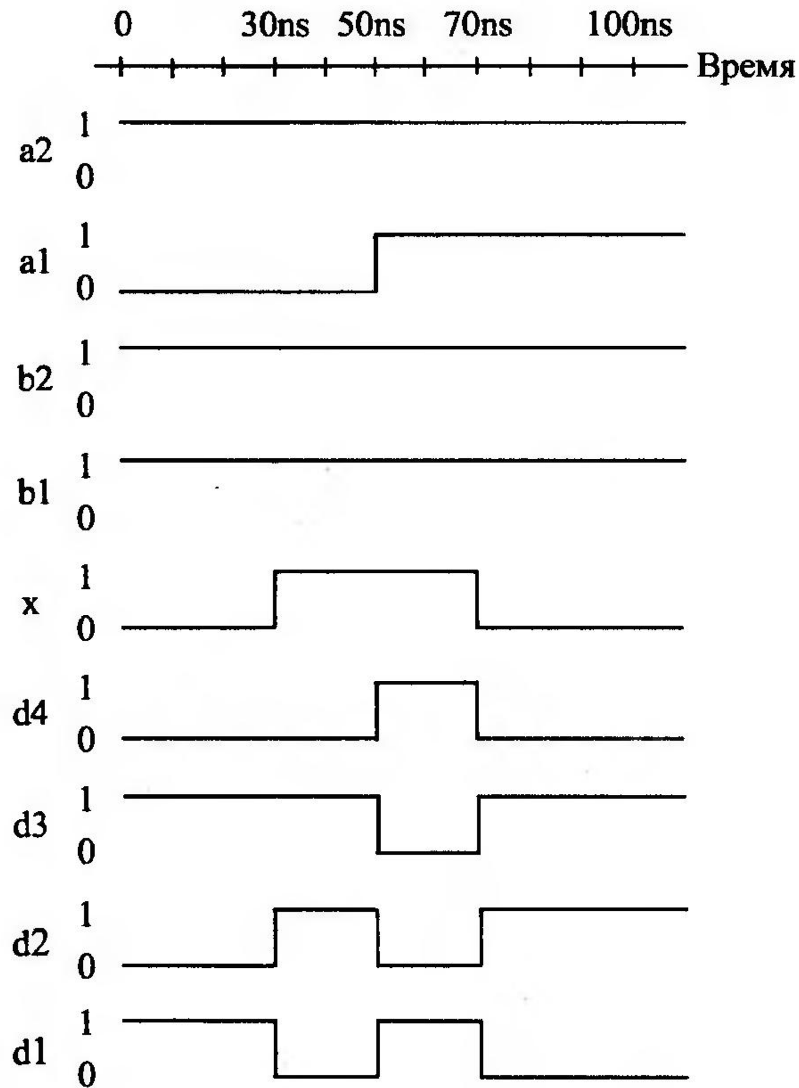

Рис. 4.12. Временная диаграмма

Таблица 4.4

Сигналы	Время			
	[0ns, 30ns]	[30ns, 50ns]	[50ns, 70ns]	[70ns,100ns]
a = (a2, a1) b = (b2, b1) x = (x)	2 3 0 (сложение)	2 3 1(умножение)	3 3 1(умножение)	3 3 0 (сложение)
d = (d4, d3, d2, d1)	5	6	9	6

В данном примере объект проекта test_vlsi_1 представляет собой «оболочку» (рис. 4.13) объекта проекта vlsi_1 с единственной целью — организовать подачу входных сигналов в объект vlsi_1, так как **внутри** архитектурного тела объекта vlsi_1 нельзя назначить входные сигналы. Входные сигналы для entity должны быть назначены **«снаружи»**!

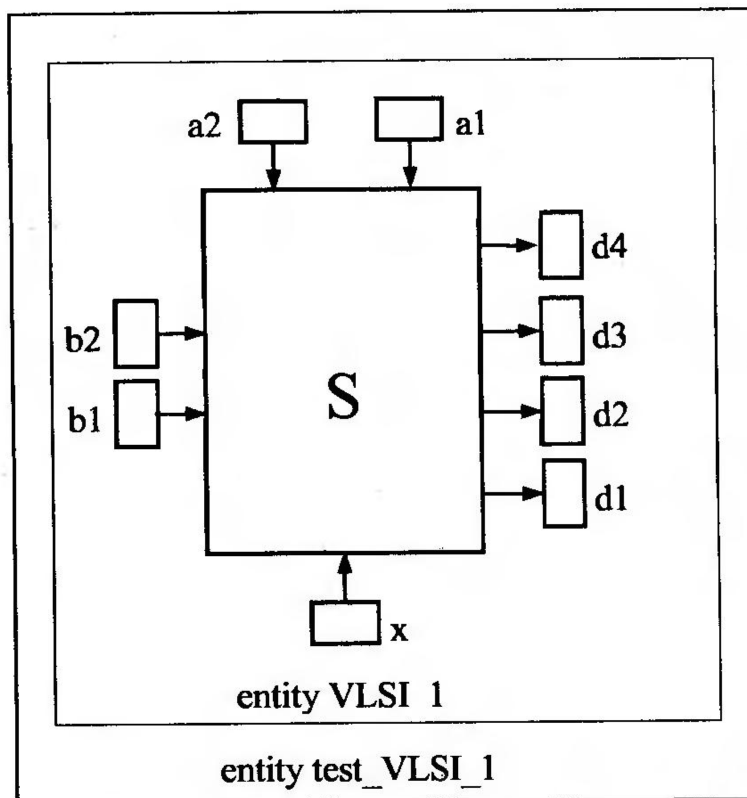

Рис. 4.13. При тестирование объекта VLSI_1 entity test_VLSI_1 не имеет портов

4.5. Синтезируемое подмножество языка VHDL

После того как проектировщик убедился в корректности VHDL-модели цифровой системы, возникает проблема программной либо схемной реализации данной модели. Схемные реализации, как правило, являются более быстродействующими, поэтому важной задачей является задача синтеза схемы, функции которой реали-

зуют поведение VHDL-модели. На практике, однако, переход к соответствующей логической схеме осуществляется не для всего языка VHDL, а только для некоторого подмножества этого языка, называемого *синтезируемым* подмножеством. Для VHDL-модели цифровой системы, описанной на синтезируемом подмножестве языка VHDL, можно получить схему. Обычно это интегральная схема типа ПЛИС либо схема программируемой пользователем вентильной матрицы [5]. Общий метод получения такой схемы является *компилятивным* — операторы языка VHDL заменяются *компилятами*. Компилят — это подсхема, реализующая вполне определенный оператор (конструкцию) языка, например оператор сложения. При синтезе схемы *все* типы данных языка VHDL заменяются типами std_logic, std_logic_vector пакета STD_LOGIC_1164. Перечислимые типы данных при синтезе *кодируются* булевыми векторами. Кодирование обычно ведется с использованием минимального числа *кодирующих переменных*. Например, для перечислимого типа

```
type my_state is (RESET, IDLE, RW_CYCLE, INT_CYCLE);
```

может быть проведено кодирование

```
RESET = "00",
IDLE = "01",
RW_CYCLE = "10",
INT_CYCLE = "11".
```

Очевидно, в кодах 00, 01, 10, 11 используются только две кодирующие булевы переменные.

Компилятивный подход — не единственный для реализации алгоритмических описаний. Например, в работе [4] предлагается способ схемной реализации параллельных процессов, описанных в терминах только двоичных переменных. Предложенный в [4] язык описания параллельных процессов и способ схемной реализации базируются на мощном формализме сетей Петри.

Переменные языка VHDL, мгновенно передающие свои значения, могут исчезать при схемной реализации. В приведенном ниже примере проследите за переменной W в VHDL-коде и в соответствующей схеме (рис. 4.14).

Пример схемной «реализации» переменных языка VHDL.

```
Library IEEE;
use IEEE.std_logic_1164.all;
entity xor_var is
port (A, B, C : in std_logic;
      X, Y : out std_logic);
end xor_var;
architecture example of xor_var is
begin
P : process (A, B, C)
variable W: std_logic;
  begin
  W := A;    X <= C xor W;
  W := B;    Y <= C xor W;
end process;
end example;
```

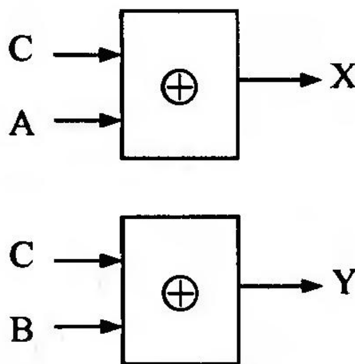

Рис. 4.14. Исчезновение переменных при схемной реализации

Для логических операторов and, or, nand, nor, xor, xnor, not, над типами bit, boolean, arrays, bit_vector, компиляция в схему осуществляется прямым преобразованием логических выражений в соответствующие логические вентили (элементы).

Пример схемной реализации логических выражений.

```
entity logical_ops_1 is
port (a, b, c, d: in bit; m: out bit);
end logical_ops_1;

architecture example of logical_ops_1 is
signal e: bit;
begin
m <= (a and b) or e;        -- оператор назначения сигнала
e <= c xor d;
 end example;
```

Схема, реализующая поведение объекта logical_ops_1, приведена на рис. 4.15.

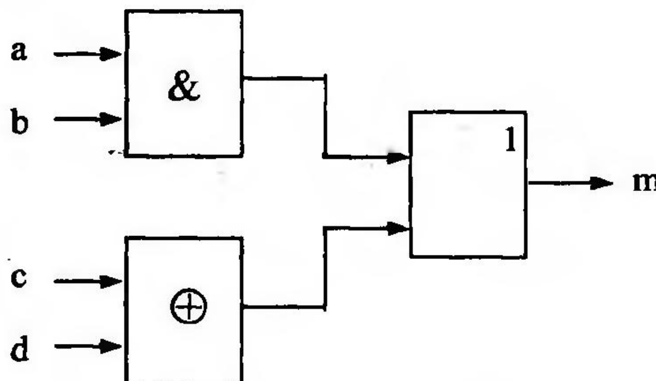

Рис. 4.15. Схемная реализация логического выражения для типа bit

Пример схемной реализации логических выражений типа bit_vector.

```
entity logical_ops_2 is
port (a, b: in bit_vector (0 to 3); m: out bit_vector (0 to 3));
end logical_ops_2;
architecture example of logical_ops_2 is
```

```
begin
m <= a and b;
end example;
```

Соответствующая схема приведена на рис. 4.16.

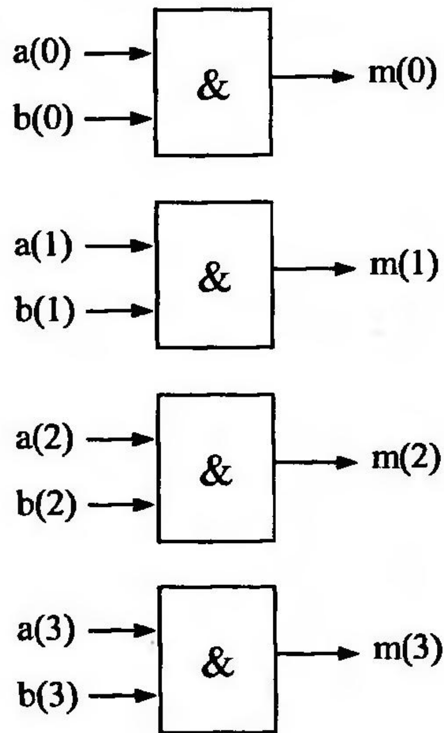

Рис. 4.16. Схемная реализация логического выражения для типа bit_vector

Схемная реализация операторов отношения

Операторами отношения являются следующие операторы = (равно); /= (не равно); > (больше); < (меньше); >= (больше или равно); <= (меньше или равно).

Результирующим типом для всех операторов сравнения является тип boolean. Операторы сравнения *не поддерживаются* при синтезе для операндов, имеющих тип real.

Следующие два примера показывают схемную реализацию для операторов a = b, a>= b.

```
entity relational_ops_1 is
port (a, b: in bit_vector (0 to 3); m: out boolean);
end relational_ops_1;
architecture example of relational_ops_1 is
begin
 m <= a = b;
end example;
```

Схема, реализующая оператор равенства для типа bit_vector, приведена на рис. 4.17.

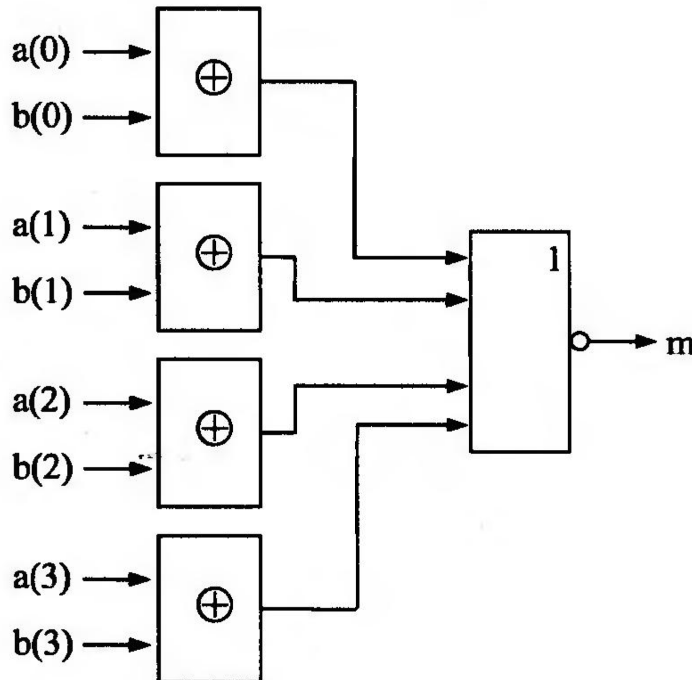

Рис. 4.17. Схемная реализация оператора a = b для типа bit_vector (0 to 3)

```
entity relational_ops_2 is
port (a, b: in integer range 0 to 3; m: out boolean);
end relational_ops_2;
architecture example of relational_ops_2 is
begin
 m <= a >= b;
end example;
```

Схема, реализующая оператор >= для типа integer, показана на рис. 4.18.

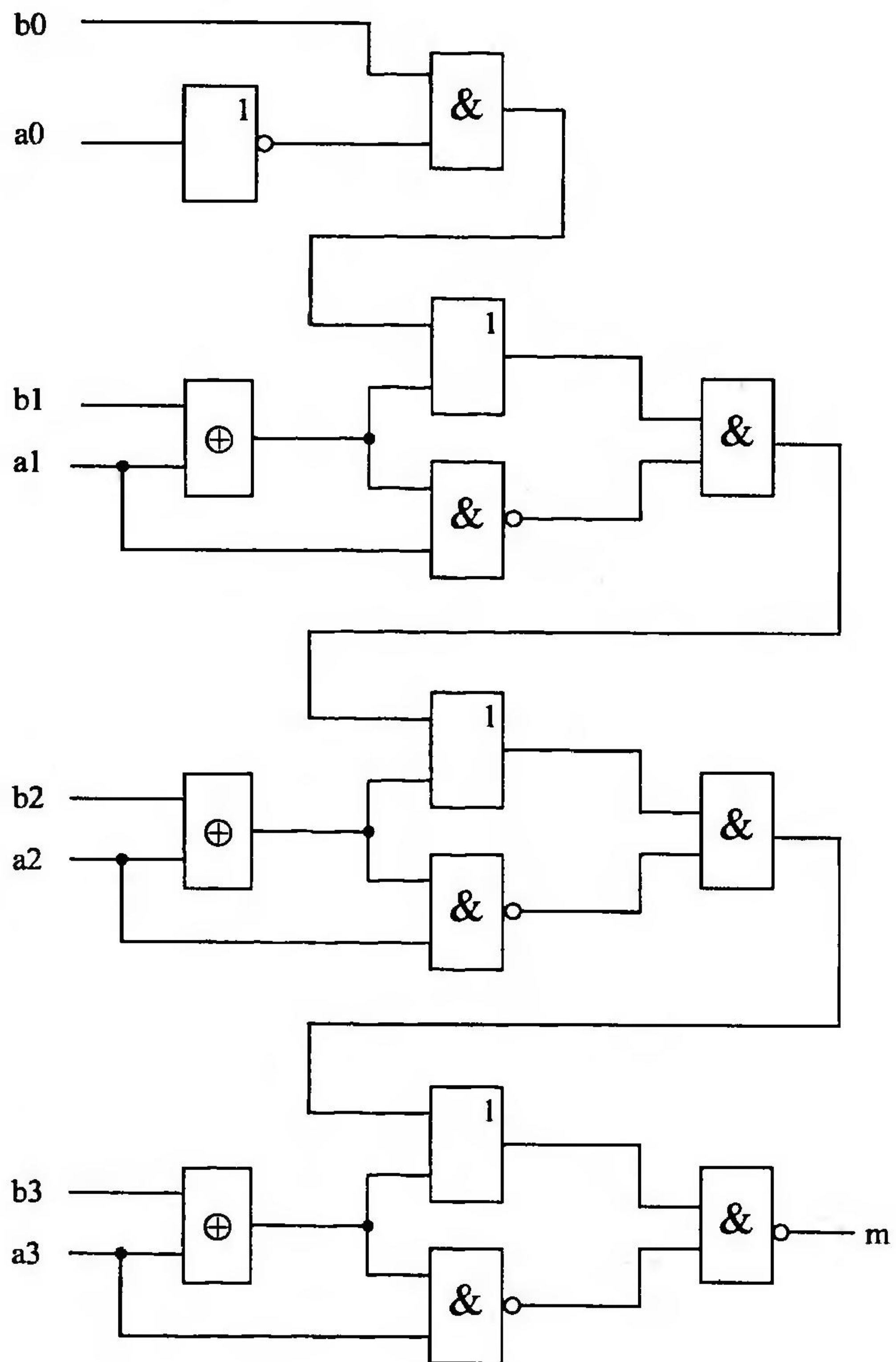

Рис. 4.18. Схемная реализация оператора отношения **a** >= **b** для целых чисел **a** = (a3, a2, a1, a0), **b** = (b3, b2, b1, b0)

Схемная реализация арифметических операторов

Арифметические операторы

+ (сложение), – (вычитание), * (умножение), / (деление), **mod** (модуль), **rem** (остаток), **abs** (абсолютное значение), ** (возведение в степень) реализуются соответствующими логическими схемами для операндов типа integer, однако не реализуются логическими схемами для операндов типа real.

! Вещественные типы данных и вещественная арифметика не поддерживаются при синтезе.

Операторы (+, –) являются достаточно «дорогостоящими» (дающими сложные логические схемы). Операторы (*, / , **mod**, **rem**) являются очень дорогостоящими. Обычно при синтезе делается специальная оптимизация при схемной реализации стандартных операторов. Реализация оператора (**abs**) не является сложной. Оператор (**) поддерживается только для констант.

Примеры схем, реализующих операторы сложения и умножения, уже рассматривались в данной книге. Например, сложение двухразрядных чисел осуществляется схемой adder_2, умножение двухразрядных чисел — схемой mult_2 (см. разд. 1.1), сложение N-разрядных чисел — схемой adder_N (см. разд. 2.2).

Схемная реализация операторов управления

Схемным «аналогом» оператора if является, по существу, мультиплексор (рис. 4.19) с одним управляющим входом, где b, c выступают в качестве настроечных входов, вход a — в качестве управляющего. Функционирование мультиплексора описывается следующей формулой: $m = \overline{a} \& c \vee a \& b$.

```
entity control_if is
 port (a, b, c: boolean; m: out boolean);
 end control_if;
architecture example of control_if is
begin
process (a, b, c)
```

```
variable n: boolean;
begin
if a then n := b;
 else n := c;
 end if;
m <= n;
 end process;
end example;
```

Схема, реализующая оператор if, изображена на рис. 4.19: если a = 1, то m = b, если a = 0, то m = c. Предлагаем читателю сравнить функцию, реализуемую процедурой MX, содержащейся в пакете multiplexer (разд.1.2), с функцией объекта проекта control_if.

Однако не всегда очевидна та логическая схема, которая соответствует оператору языка VHDL. В качестве примера укажем на тот факт, что в компиляторах, осуществляющих схемную реализацию VHDL-кода, оператор if, у которого после ключевого слова else не производится никаких действий, интерпретируется как D-триггер (задержка). Поэтому получение нужных логических схем по VHDL-кодам с помощью САПР требует тщательного изучения библиотеки компилятов.

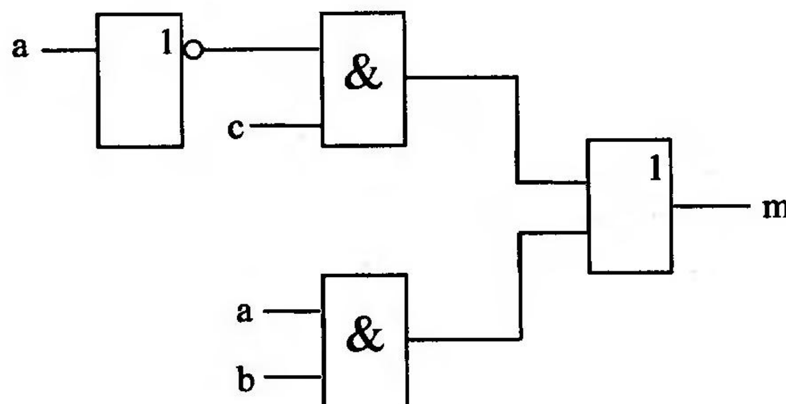

Рис. 4.19. Схемная реализация оператора управления if (entity control_if)

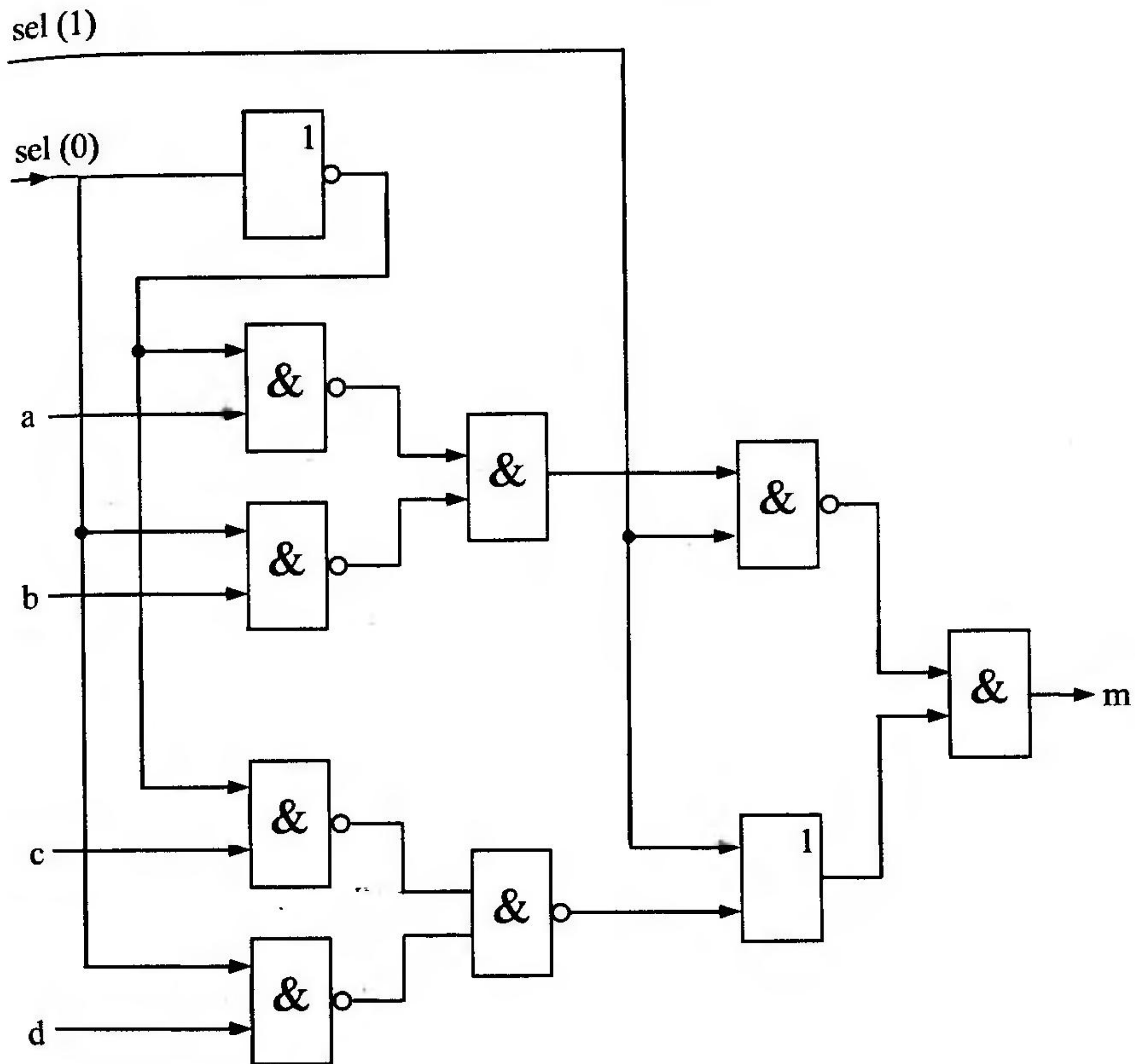

Рис.4.20. Схемная реализация оператора case (entity control_case)

Схемная реализация оператора case

entity control_case **is**
port (sel: **in** bit_vector (0 **to** 1); a, b, c, d : **in** bit; m: **out** bit);
end control_case;
architecture example **of** control_case **is**
begin
process (sel, a, b, c, d)

```
begin
case sel is
        when      "00" => m <= c;
        when      "01" => m <= a;
        when      "10" => m <= d;
        when others  => m <= b;
end case;
end process;
end example;
```

Схема, реализующая оператор case, изображена на рис. 4.20.

Схемная реализация подпрограмм и циклов

Операторы generate, loop (for loop, while loop), function, procedure также могут быть схемно реализованы.

Для каждого вызова подпрограммы и для каждой итерации цикла создается соответствующая схема.

```
entity control_loop is
port (a: bit_vector (0 to 3); m: out bit_vector (0 to 3));
end control_loop;
architecture example of control_loop is
begin
        process (a)
                variable b: bit;
        begin
        b := '1';
                for i in 0 to 3 loop
            b:= a(3-i) and b;
            m(i) <= b;
                end loop;
        end process;
end example;
entity subprograms is
port (a: bit_vector (0 to 2); m: out bit_vector (0 to 2));
 end subprograms;
```

```
architecture example of subprograms is
 function simple (w, x, y: bit)
return bit is
begin
 return (w and x) or y;
 end;
begin
 process (a)
begin
 m(0) <= simple(a(0), a(1), a(2));
 m(1) <= simple(a(2), a(0), a(1));
 m(2) <= simple(a(1), a(2), a(0));
end process;
end example;
```

Например, для цикла (entity loop_stmt) и для подпрограммы (entity subprograms) соответствующие схемы приведены ниже (рис. 4.21, 4.22).

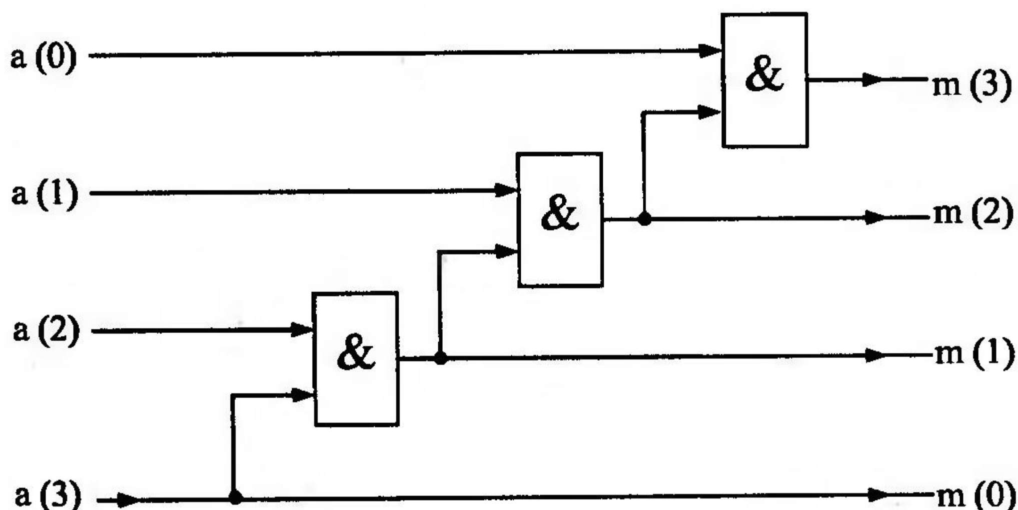

Рис. 4.21. Схемная реализация оператора loop (entity control_loop)

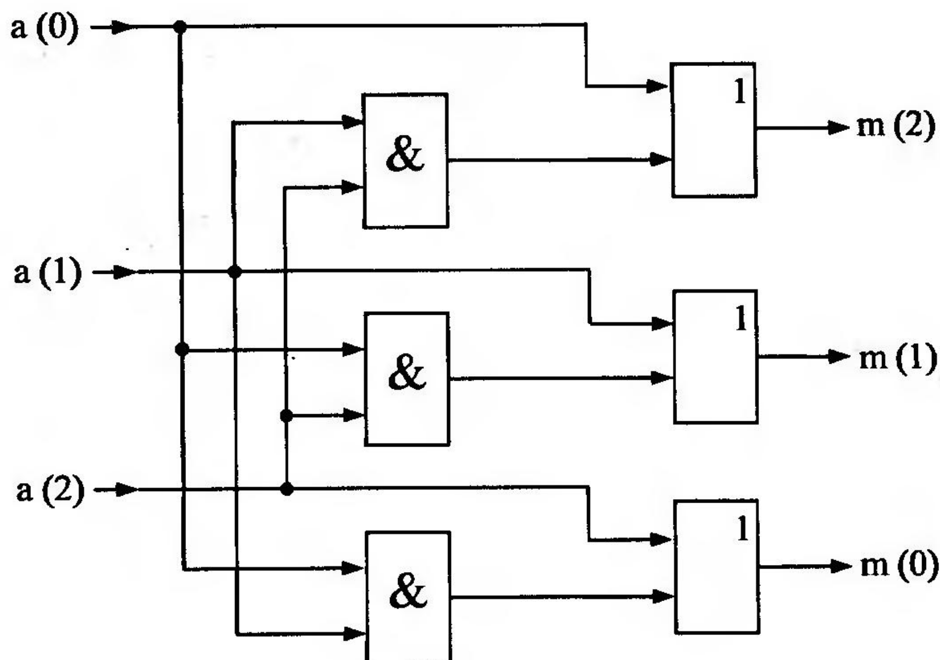

Рис. 4.22. Схемная реализация оператора «подпрограмма» (entity subprograms)

В заключение можно сказать, что мир VHDL большой и достаточно сложный, так как сложны те электронные схемы и системы, инструментом описания, моделирования, верификации и синтеза которых он является. За чертой остались еще такие аспекты языка VHDL, как типы файлов и типы доступа, инородные подпрограммы, группы, разделяемые переменные и другие вопросы. Автор надеется, что провести их изучение можно будет, базируясь на материалах, представленных в данной книге.

УПРАЖНЕНИЯ

1. Правильны ли утверждения

a) структурное VHDL-описание может быть иерархичным;

b) поведенческое VHDL-описание может быть иерархичным;

c) смешанное (структурно-поведенческое) VHDL-описание не может быть иерархичным.

2. Правильно ли, что структурное описание состоит из компонент и сигналов?

3. Правильно ли, что все компоненты должны быть специфицированы на поведенческом уровне?

4. Какие компоненты проекта не имеют структурного описания?

5. Какое устройство реализует поведение, представленное VHDL-кодом?

```
Library IEEE;    use IEEE.std_logic_1164.all;
entity EX is
port (A : in std_ulogic_vector(0 to 15);
    SEL : in integer range 0 to 15;
    Z   : out std_ulogic);
end EX;
architecture RTL of EX is
begin
  WHAT: process (A, SEL)
  begin
    for I in 0 to 15 loop
      if SEL = I then
        Z <= A(I);
      end if;
    end loop;
  end process WHAT;
end RTL;
```

Выберите правильный ответ.

а) дешифратор;
в) счетчик;
с) мультиплексор.

Укажите разрядность устройства.

6. Опишите поведение конечного автомата, используя двухразмерные массивы для задания совмещенной таблицы переходов и выходов.

7. Используя информацию о структурной схеме, приведенную в [2, с. 20–21], составьте VHDL-код и проведите моделирование конечного автомата, поведение которого задано в архитектурном теле rtl_a (entity FSM_A).

8. Составьте VHDL-код для генерации синхроимпульсов (см. рис. 4.5) в виде процесса с использованием оператора wait.

9. Как должен быть написан VHDL-код, чтобы с помощью существующих средств САПР можно было автоматически получить схему? Выберите правильный ответ.

a) код должен быть написан на поведенческом уровне;

b) код должен быть написан на уровне регистровых передач;

c) код должен быть написан на уровне логических вентилей;

d) средства САПР могут синтезировать схему для любого уровня VHDL-описания.

10. В данной книге даны различные архитектурные тела для D-триггера. Рассмотрите особенности их функционального и структурного описания. Проведите моделирование.