Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных средств

М. И. Вашкевич, В. Н. Пригодич

ПРОЕКТИРОВАНИЕ ЦИФРОВЫХ УСТРОЙСТВ НА ЯЗЫКАХ VHDL И SYSTEMVERILOG. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве пособия для специальности 6-05-0611-05 «Компьютерная инженерия»

Минск БГУИР 2025

УДК 004.43(076.5)
ББК 32.973.2я73
В23

Рецензенты:

кафедра «Технология и методика преподавания»
Белорусского национального технического университета
(протокол № 1 от 02.09.2024);

ведущий инженер-программист ООО «МайкроДизайн»
кандидат физико-математических наук, доцент С. Е. Бухтояров

Вашкевич, М. И.

В23 Проектирование цифровых устройств на языках VHDL и SystemVerilog. Лабораторный практикум : пособие / М. И. Вашкевич, В. Н. Пригодич. – Минск : БГУИР, 2025. – 91 с. : ил.

ISBN 978-985-543-806-0.

Предназначено для изучения основ проектирования цифровых устройств с использованием языков описания аппаратуры – VHDL (Very high speed integrated circuit Hardware Description Language) и SystemVerilog. Подробно разобраны основные конструкции языка VHDL, используемые для описания комбинационных схем и тестового окружения VHDL-проекта. Уделено внимание вопросам иерархического и поведенческого описания работы цифровых устройств. Описан процесс моделирования цифровых устройств с учетом временных задержек логических вентилей, показано, как оценить критический путь комбинационной схемы. Дано описание основных средств языка SystemVerilog, используемых как для структурного описания проектов, так и описания на уровне регистровых передач. Приведен общий маршрут проектирования цифрового устройства в САПР Vivado.

УДК 004.43(076.5)
ББК 32.973.2я73

ISBN 978-985-543-806-0

СОДЕРЖАНИЕ

ЛАБОРАТОРНАЯ РАБОТА № 1. ПРОЕКТИРОВАНИЕ КОМБИНАЦИОННЫХ СХЕМ НА VHDL

ЦЕЛЬ РАБОТЫ: изучение основных конструкций языка VHDL, освоение маршрута проектирования и моделирования цифрового устройства в САПР Vivado.

1.1. Теоретические сведения

1.1.1. Краткая история VHDL

Язык VHDL (*Very high speed integrated circuits Hardware Description Language*) применяется для описания цифровых устройств во многих САПР, используемых для разработки заказных интегральных схем (*application specific integrated circuit – ASIC*), а также для создания проектов на базе программируемых логических интегральных схем (ПЛИС) и программируемых пользователями вентильных матриц (*Field-Programmable Gate Arrays – FPGA*). Данный язык предназначен для описания проектируемых систем и их моделирования на начальных этапах проектирования – алгоритмическом и логическом.

VHDL был разработан по заказу министерства обороны США. Первоначально язык VHDL предназначался для описания структуры и функциональности электронных схем и выступал альтернативным способом представления принципиальной и функциональной схемы устройства. Однако вскоре стало понятно, что такой способ описания хорошо подходит для задач моделирования, а также синтеза цифровых схем. В 1987 году конструкции языка VHDL были закреплены в стандарте IEEE 1076–1987. Данный стандарт еще называют VHDL’87. В 1993 году был принят второй стандарт на язык VHDL – IEEE 1076–1993, называемый еще VHDL’93. Опыт разработчиков со временем обобщается и формализуется в новых версиях стандартов на языки описания аппаратуры. В настоящее время наибольшее распространение имеет версия языка VHDL 2008 года (стандарт IEEE 1076–2008).

1.1.2. Базовые конструкции языка VHDL

Чтобы начать знакомство с языком VHDL, рассмотрим, как на нем можно описать двухвходовой вентиль И (рис. 1.1).

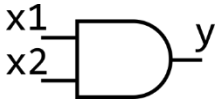

Рис. 1.1. Вентиль И

Ниже приведено VHDL-описание вентиля И.

```
entity gate_and is
  port (
    x1 : in  std_logic;
    x2 : in  std_logic;
    y  : out std_logic
  );
end gate_and;

architecture RTL of gate_and is
begin
  y <= x1 and x2;
end RTL;
```

Рассмотрев внимательно данный код, можно отметить ряд особенностей VHDL-описания цифровых схем. Описание начинается с ключевого слова `entity`, после которого указывается название схемы, далее описывается ее интерфейс. Таким образом, `entity` определяет интерфейс между объектом и внешним окружением, при этом «внутренность» объекта проекта уподобляется «черному ящику».

В интерфейсе рассматриваемой схемы присутствует несколько портов, которые задаются с использованием следующего формата.

```
<идентификаторы портов> : <направление> <тип порта>;
```

В приведенном примере схема имеет два входных порта: `x1` и `x2`, а также один выходной порт `y`. Для каждого порта указывается его направление `in` или `out`, кроме этого, для описания двунаправленных портов используется ключевое слово `inout`. Также для портов указывается тип данных, в данном случае это `std_logic`. Этот тип данных используется для описания логических уровней цифровых сигналов. Помимо значений `‘0’` и `‘1’` тип `std_logic` может принимать значения `‘Z’` (третье состояние), `‘X’` (неопределенное состояние) и некоторые другие.

После интерфейсной части схемы описывается ее архитектурное тело (рис. 1.2).

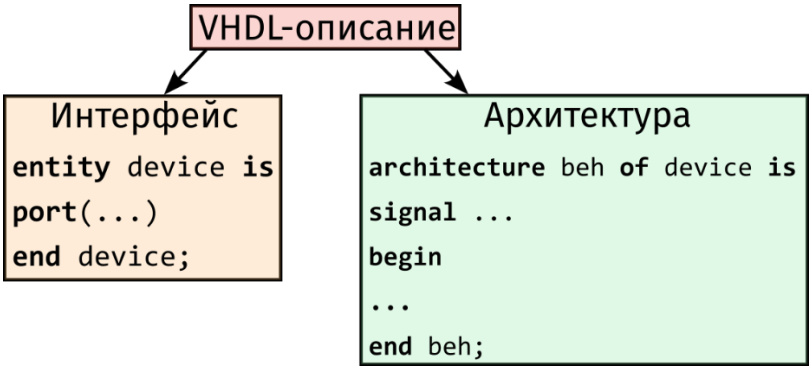

Рис. 1.2. Структура VHDL-описания

В приведенном примере описания вентиля И архитектурное тело имеет название RTL. Внутри архитектурного тела описывается логика работы схемы. В рассматриваемом примере выходу y назначается результат выполнения логической операции И над двумя входными сигналами x1 и x2 (рис. 1.3).

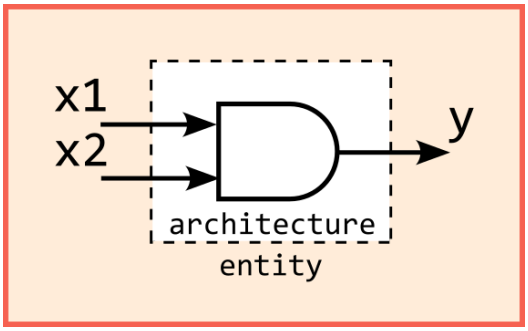

Рис. 1.3. VHDL-описание: интерфейс (entity) и архитектура

В общем виде интерфейс схемы/устройства на языке VHDL описывается следующим образом.

```
entity <название схемы> is
[generic(<список параметров>);]
[port(<список портов>);]
end [<название схемы >];
```

Помимо уже известной нам секции port в интерфейсе может содержаться секция generic, которая используется для задания параметров схемы. Например, при описании сумматора в качестве параметра может выступать разрядность входных/выходных сигналов.

Общее описание архитектурного тела приведено ниже.

```
architecture <имя архитектуры> of <название схемы> is
[<область объявлений архитектуры>]
begin
  <операторы архитектуры>
end [<имя архитектуры>];
```

Язык VHDL позволяет для каждого объекта иметь несколько архитектур. Например, часто для одного объекта создают два описания архитектуры: одно поведенческое, а другое структурное. «Внутренность» архитектуры представляет собой ряд параллельных операторов.

VHDL-описание устройства, состоящее из интерфейса (entity) и архитектуры (architecture), сохраняется в проектном файле с расширением .vhd (рис. 1.4).

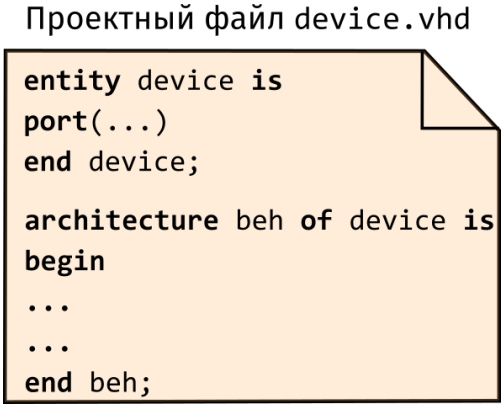

Рис. 1.4. Проектный файл с VHDL-описанием устройства

VHDL является языком, нечувствительным к регистру, поэтому объекты с именами `Temp1` и `TEMP1` будут считаться одним и тем же объектом.

VHDL является языком со строгой типизацией. В частности, это означает, что при попытке присвоить сигналу значение, которое не имеет полного соответствия с типом этого сигнала, компилятор языка будет выдавать ошибку. У данного подхода есть и преимущества, и недостатки. С одной стороны, строгая типизация позволяет не делать глупых ошибок, а с другой – на разработчика ложится дополнительный объем работы, заключающийся в явном приведении типов даже в случаях, которые кажутся очевидными.

1.1.3. Структурное и поведенческое описание цифровой системы

Структурное описание – это описание системы в виде совокупности компонентов и связей между ними. Поведенческое описание – описание системы на уровне зависимости выходов от входов при помощи некоторых процедур. Таким образом, поведенческое описание задает алгоритм, реализуемый системой.

Рассмотрим отличие структурного и поведенческого описания цифровой схемы на примере схемы, показанной на рис. 1.5.

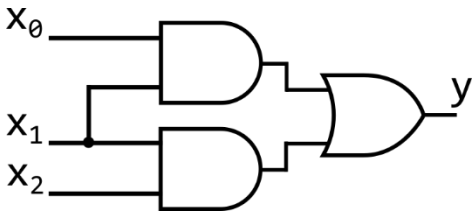

Рис. 1.5. Комбинационная схема

Чтобы составить для данной схемы структурное VHDL-описание, обозначим на ней внутренние сигналы, как показано на рис. 1.6.

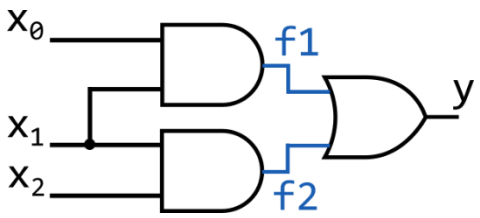

Рис. 1.6. Комбинационная схема с обозначением внутренних сигналов

VHDL-описание данной схемы показано в листинге 1.1.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity device_01 is
  port (x: in std_logic_vector(2 downto 0);
        y: out std_logic);
end device_01;

architecture Structural of device_01 is
signal f1, f2 : std_logic;
begin

f1 <= x(0) and x(1);
f2 <= x(0) and x(2);
y <= f1 or f2;

end Structural;
```

Листинг 1.1. VHDL-описание комбинационной схемы

Разберем особенности данного VHDL-описания. Первые две строки нужны, чтобы подключить библиотеку `IEEE.STD_LOGIC_1164`, в которой определен тип `std_logic`. Далее следует описание интерфейса схемы, начинающееся с ключевого слова `entity`. В отличие от предыдущего примера при объявлении входов схемы использован тип `std_logic_vector`, который также содержится в библиотеке `STD_LOGIC_1164`. Как следует из названия, тип `std_logic_vector` является векторным типом данных, причем каждый элемент вектора имеет тип `std_logic`. Особенностью `std_logic_vector` является возможность восходящей (`to`) и нисходящей (`downto`) индексации элементов вектора. В разбираемом примере объявлен трехэлементный вектор, который имеет нисходящую индексацию элементов, как показано на рис. 1.7.

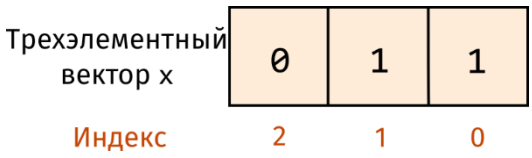

Рис. 1.7. Вектор с нисходящей индексацией – `std_logic_vector(2 downto 0)`

Таким образом, при использовании нисходящей индексации первый элемент вектора имеет максимальное значение индекса. Восходящая индексация более привычна и напоминает использование обычных одномерных массивов в языках программирования высокого уровня. Пример восходящей индексации показан на рис. 1.8.

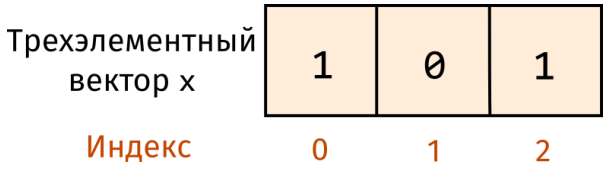

Рис. 1.8. Вектор с восходящей индексацией – `std_logic_vector(0 to 2)`

Можно сделать вывод, что тип `std_logic_vector` – это простой способ представления группы сигналов или шины данных. Наличие двух способов индексации обусловлено тем, что в цифровых устройствах данные могут храниться в различном порядке: иногда начальные биты являются старшими (в этом случае удобнее использовать нисходящую индексацию), а иногда начальные биты являются младшими, и в этом случае лучше подойдет восходящая индексация. Нисходящий порядок разрядов называется *little-endian* (т. е. «оканчивающийся на младший»). Восходящая индексация описывает порядок бит, называемый *big-endian* (т. е. «оканчивающийся на старший»), поскольку младший бит имеет наибольшее значение индекса.

Вернемся к рассматриваемому примеру (см. листинг 1.1). После объявления для `device_01` архитектуры с именем `Structural` следует объявление внутренних сигналов схемы, которое начинается с ключевого слова `signal`. В данном случае внутренними являются два сигнала с именами `f1` и `f2`, имеющие тип `std_logic`. Непосредственно схема, показанная на рис. 1.6, описывается тремя строками VHDL-кода.

```
f1 <= x(0) and x(1);
f2 <= x(0) and x(2);
y <= f1 or f2;
```

Каждая строка, по сути, отражает работу одного логического вентиля. Таким образом, полученное VHDL-описание можно считать примером структурного описания, поскольку операторы языка использованы для описания структуры цифровой схемы. Список логических операций языка VHDL шире, чем использованные выше `and` и `or`, полный их перечень приведен в табл. 1.1.

VHDL не определяет соотношение приоритетов логических операций (`and`, `or` и т. д.), поэтому при записи логических выражений нужно всегда использовать скобки.

Таблица 1.1

Логические операторы VHDL

Операция	Значение
and	Операция И
or	Операция ИЛИ
nand	Операция И-НЕ
nor	Операция ИЛИ-НЕ
xor	Исключающее ИЛИ
xnor	Исключающее ИЛИ-НЕ

Рассматриваемая схема (см. рис. 1.6) может также быть описана с использованием поведенческого стиля. Ниже приведен вариант такого описания.

```
entity device_01 is
  port (x: in std_logic_vector(2 downto 0);
        y: out std_logic);
end device_01;

architecture Behavioral of device_01 is
begin

process (x) begin
    if (x(0)='1') and (x(1)='1') then
        y <= '1';
    elsif (x(0)='1') and (x(2)='1') then
        y <= '1';
    else
        y <= '0';
    end if;
end process;

end Behavioral;
```

Перед тем как приступить к анализу приведенного VHDL-описания, следует заметить, что здесь использованы операторы языка, которые ранее не упоминались. В частности, речь идет об операторе process. Формально данный оператор имеет следующий синтаксис.

```
<имя процесса>: process([<список сигналов активации>])
[<область объявления>]
begin
<последовательные операторы>
end process <имя процесса>;
```

Процесс является языковой конструкцией языка VHDL, которая используется для описания функциональных блоков, работающих параллельно. Концепция процесса полностью соответствует логике работы цифровых устройств, в которых внешние сигналы могут поступать сразу на несколько внутренних блоков и обрабатываться параллельно. При помощи оператора `process` можно описывать как комбинационные, так и последовательностные схемы.

Как видно из описания, после ключевого слова `process` указывается список сигналов, которые переводят данный процесс в активное состояние. Часто его называют *списком чувствительности* процесса. Важно уяснить, что несмотря на то, что процесс сам по себе является параллельным оператором, действия внутри процесса выполняются последовательно. То есть внутри процесса описывается логика получения выходных сигналов после того, как входные сигналы из списка чувствительности изменились.

Вернемся к архитектуре `Behavioral` модуля `device_01`. Вся логика работы устройства описана внутри одного процесса. В качестве списка чувствительности указан вектор `x`, поскольку выход схемы `y` зависит от всех компонентов вектора. В первой ветке условного оператора проверяется одновременное равенство единице сигналов `x(0)` и `x(1)`, во второй ветке проверяется на равенство единице `x(0)` и `x(2)`. В обоих случаях выходу схемы `y` присваивается значение единицы. Данные условия можно было проверить, используя только одну ветку оператора `if`, однако это снизило бы наглядность кода. Если условия установки `y` в единицу не выполняются, то `y` устанавливается в нуль.

Таким образом, поведенческое описание лучше выражает логику работы устройства, чем структурное, и поэтому чаще применяется на практике.

1.1.4. Иерархическое описание проекта

Важной особенностью VHDL является возможность иерархического описания проекта. Это означает, что можно составить VHDL-описание отдельных блоков (модулей) проекта, а затем объединить их в схему верхнего уровня. Для вставки модуля нижнего уровня в модуль верхнего уровня используется языковая конструкция создания экземпляра (англ. *instantiation*). На рис. 1.9 показана иерархическая структура проекта из нескольких модулей.

Рассмотрим опять устройство, показанное на рис. 1.6. В его состав входят вентили И и ИЛИ, которые можно описать в виде VHDL-модулей, как показано в листинге 1.2.

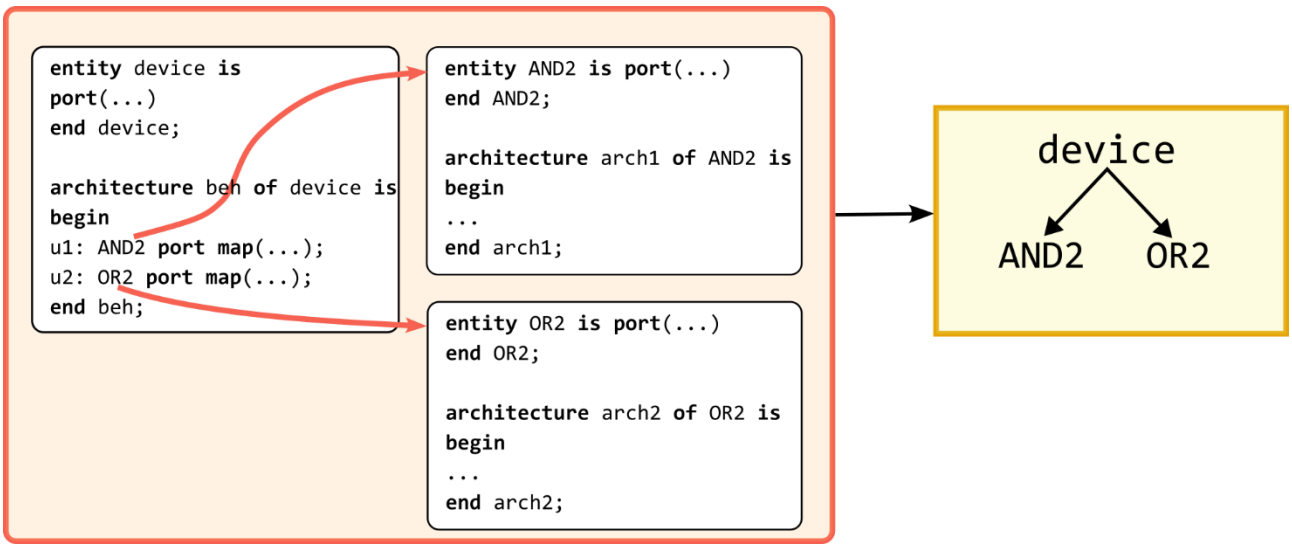

Рис. 1.9. Иерархическое описание VHDL-проекта

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND2 is
    Port (a : in STD_LOGIC;
          b : in STD_LOGIC;
          c : out STD_LOGIC);
end AND2;

architecture arch1 of AND2 is
begin
c <= a and b;
end arch1;
```

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity OR2 is
    Port (a : in STD_LOGIC;
          b : in STD_LOGIC;
          c : out STD_LOGIC);
end OR2;

architecture arch2 of OR2 is
begin
c <= a or b;
end arch2;
```

Листинг 1.2. VHDL-описание базовых вентилей И и ИЛИ

Дополним схему на рис. 1.6, введя в нее обозначение вентилей и их метки (рис. 1.10).

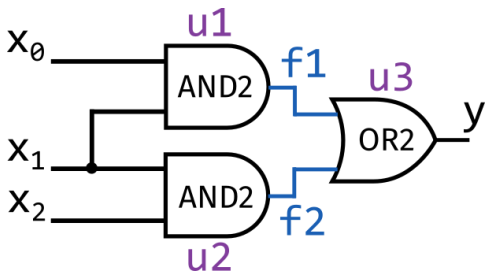

Рис. 1.10. Схема устройства с обозначением связей и модулей

Ниже приводится VHDL-описание схемы, представленной на рис. 1.10.

```
entity device_01 is
  port (x: in std_logic_vector(2 downto 0);
        y: out std_logic);
end device_01;

architecture Beh3 of device_01 is
component AND2 port (a : in STD_LOGIC;
           b : in STD_LOGIC;
           c : out STD_LOGIC);
end component;

component OR2 port (a : in STD_LOGIC;
           b : in STD_LOGIC;
           c : out STD_LOGIC);
end component;

signal f1, f2 : std_logic;
begin
u1: AND2 port map(a=>x(0), b=>x(1), c=>f1);
u2: AND2 port map(a=>x(0), b=>x(2), c=>f2);
u3: OR2 port map(a=>f1, b=>f2, c=>y);
end Beh3;
```

В отличие от предыдущих примеров в данном VHDL-коде внутри архитектуры имеется декларация компонентов, которые впоследствии будут использованы. В нашем случае объявляются компоненты AND2 и OR2, которые были описаны ранее. Файлы с описанием компонентов должны находиться в одном каталоге с описанием модуля верхнего уровня. Компоненты, используемые при описании архитектуры, должны быть описаны в области объявления архитектуры. В общем случае описание компонента имеет следующий вид.

```
component <имя модуля>
  generic(<список параметров>);
  port(<список портов>);
end <имя модуля>;
```

Список параметров generic и список портов port должны соответствовать описаниям generic и port соответствующего модуля.

Для использования компонента внутри архитектуры ему необходимо присвоить уникальное имя и назначить портам компонента сигналы. Создание экземпляра компонента осуществляется оператором port map, который имеет следующий синтаксис.

```
<уникальное имя компонента> : <имя интерфейса>
port map (<список сигналов>);
```

Список сигналов связывает сигналы описываемого устройства с портами компонента. В рассматриваемом примере экземпляр вентиля OR2 создается следующим образом.

```
u3: OR2 port map(a=>f1, b=>f2, c=>y);
```

Здесь u3 – это уникальное имя экземпляра компонента внутри проекта.

Есть два варианта описания списка сигналов в операторе port map: позиционное соответствие и явное соответствие. Рассмотрим их на примере компонента u1:

1) явное соответствие.

```
u1: AND2 port map(a=>x(0), b=>x(1), c=>f1);
```

При явном соответствии порт компонента связывается с сигналом с помощью оператора =>, при этом назначать сигналы можно в произвольном порядке, как показано в следующем примере.

```
u1: AND2 port map(c=>f1, b=>x(1), a=>x(0));
```

2) позиционное соответствие означает, что сигналы назначаются в том порядке, в котором они были объявлены в интерфейсе схемы. Пример позиционного соответствия приведен ниже.

```
u1: AND2 port map(x(0), x(1), f1);
```

В данном примере сигнал x(0) путем позиционного соответствия будет связан с портом a, x(1) с портом b, а f1 с портом c. Обычно разработчики советуют не использовать позиционное назначение, поскольку данный способ потенциально может приводить к трудно выявляемым ошибкам. Представьте себе, что в процессе проектирования вы изменили порядок объявления портом в схеме нижнего уровня. В этом случае ошибки в модуле верхнего уровня не возникнет, хотя логика работы устройства может стать неверной.

Непосредственное создание экземпляра

Помимо описанного выше, есть упрощенный вариант создания иерархического VHDL-описания проекта – через непосредственное указание интерфейса (entity) и имени архитектуры компонентов схемы. Пример такого стиля описания схемы, изображенной на рис. 1.10, приведен ниже (листинг 1.3).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity device_01 is
  port (x: in std_logic_vector(2 downto 0);
        y: out std_logic);
end device_01;

architecture Beh4 of device_01 is
signal f1, f2 : std_logic;
begin
u1: entity work.AND2(arch1) port map(a=>x(0), b=>x(1), c=>f1);
u2: entity work.AND2(arch1) port map(a=>x(0), b=>x(2), c=>f2);
u3: entity work.OR2(arch2) port map(a=>f1, b=>f2, c=>y);
end Beh4;
```

Листинг 1.3. VHDL-описание схемы устройства с использованием механизма непосредственного создания экземпляров модулей

Данное описание отличается в первую очередь тем, что здесь нет секции с описанием компонентов схемы (ключевое слово `component` не используется). При создании экземпляра каждого вентиля мы указываем его метку, после чего используем ключевое слово `entity`, чтобы показать, что мы хотим использовать *непосредственное создание экземпляра*. После этого мы специфицируем модуль, указывая библиотеку и название интерфейса и архитектуры. Пример в следующем фрагменте.

```
u1: entity work.AND2(arch1) port map(a=>x(0), b=>x(1), c=>f1);
```

Здесь `work` – это название библиотеки (в данном случае это текущая рабочая библиотека проекта); `AND2` – название интерфейса схемы; `arch1` – название архитектуры. Указывать название архитектуры необязательно, но ее указание может помочь избежать досадных ошибок в тех случаях, когда для одного интерфейса есть несколько архитектур. Если явно не указать конкретную архитектуру, то в проекте будет использоваться последняя скомпилированная архитектура.

Сравнение компонентного и непосредственного создания экземпляра объекта

Описание компонента полностью соответствует интерфейсу (`entity`) модуля, экземпляр которого вы хотите создать. Если в ходе разработки возникнет необходимость изменить интерфейс модуля, нужно будет внести изменения в трех местах проекта: в интерфейсе модуля, в объявлении компонента и в экземпляре.

Кроме того, при использовании компонентного способа создания экземпляра нет простого способа указать, какую именно архитектуру необходимо использовать. Если в проекте есть две или более архитектуры для данного модуля, то будет задействована последняя скомпилированная архитектура.

По этим причинам компонентный способ создания экземпляра модуля уступает непосредственному способу. Тем не менее существует один очень важный случай, когда компонентный способ необходим. Он используется в том случае, когда проект имеет смешанное описание (например, используются два языка: VHDL и Verilog). В этом случае, если модуль, который необходимо подключить, написан на Verilog, сделать это можно, только используя компонентный способ создания экземпляра объекта.

1.1.5. Тестовое окружение VHDL-проекта

Важной частью процесса разработки цифрового устройства является его тестирование. Полученные в процессе проектирования VHDL-описания требуют проверки правильности описываемой ими логики работы устройства.

Чтобы проверить корректность VHDL-описания некоторой цифровой схемы на языке VHDL, обычно проводят моделирование и получают временные диаграммы для всех сигналов схемы. Задачей разработчика в данном случае становится написание так называемого *тестового окружения* (англ. *testbench*). Под тестовым окружением понимается VHDL-код, который позволяет сгенерировать сигналы, необходимые для проверки корректности полученного VHDL-описания устройства. При написании теста может быть использовано как синтезируемое, так и несинтезируемое подмножество языка VHDL, поскольку тест не предназначен для отображения на аппаратуру.

Следующая диаграмма (рис. 1.11) отражает типичную структуру простого тестового окружения.

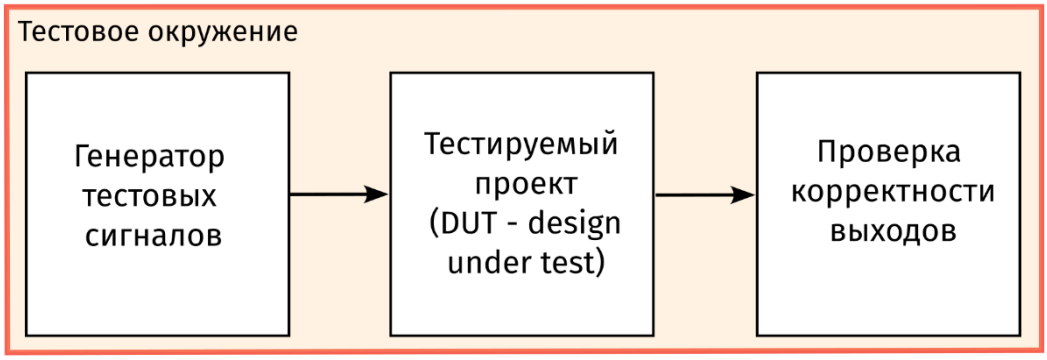

Рис. 1.11. Структура тестового окружения

Блок генератора тестовых сигналов формирует входные данные для тестируемого проекта, а отдельный блок проверяет корректность выходных данных.

Для крупных проектов генератор тестовых сигналов и средство проверки выходных данных могут находиться в отдельных файлах. Для небольших проектов разумно описывать все эти блоки в одном файле.

Блок проверки корректности вывода необходим в тех случаях, когда на передний план выходит автоматизация процесса тестирования крупных проектов. В более простом варианте он не используется, поскольку добавляет ненужной сложности. Вместо этого можно воспользоваться программой-симулятором, которая позволяет непосредственно просматривать временную форму входных/выходных сигналов. Например, в САПР Vivado (Xilinx) есть встроенное средство построения временных диаграмм сигналов.

Временное моделирование

Для моделирования процессов, происходящих во времени, в VHDL используется встроенный тип данных `time`.

```
type time is range 0 to 1E20
 units
  fs;
  ps = 1000 fs;
  ns = 1000 ps;
  us = 1000 ns;
  ms = 1000 us;
  s = 1000 ms;
  min = 60 s;
  hr = 60 min;
end units;
```

Определяя в тесте временной интервал с использованием типа `time`, необходимо указать как значение, так и единицу времени. По умолчанию минимальное временное разрешение равно 1 fs (одна фемтосекунда, или 10^{-15} с).

Ключевое слово `after` используется в VHDL для присвоения сигналам значения в указанное время в будущем. Оператор `after` можно использовать в параллельных операторах или в процессах.

Приведенный ниже фрагмент кода на VHDL показывает синтаксис оператора `after`.

```
<signal> <= [<initial_value>,] <end_value> after <time>;
```

Первая часть назначения, которая стоит перед запятой, относительно проста для понимания, поскольку `<signal> <= <initial_value>` соответствует обычному назначению сигналов. Вторая часть кода представляет собой планирование дальнейшего обновления сигнала (значением `<end_value>`) спустя указанный интервал времени (`<time>`). Переменная `<time>` должна иметь тип `time`, который был описан

выше. Оператор `after` часто используется в VHDL для сброса ПЛИС в начале моделирования, как показано в следующем примере.

```
reset <= '1', '0' after 1 us;
```

В начальный момент времени сигнал `reset` переводится в активное состояние (`'1'`). Далее при помощи оператора `after` выполняется планирование (англ. *schedule*) перехода сигнала в неактивное состояние (`'0'`) после 1 мкс модельного времени.

Также можно использовать оператор `after` без задания начального значения. Этот подход используется для планирования постоянных изменений состояния сигнала. Это полезно при генерации, например, тактовых сигналов, поскольку сигнал может инвертироваться с фиксированными интервалами. В приведенном ниже фрагменте кода показан базовый метод генерации тактовых сигналов в VHDL-тесте.

```
clock <= not clock after 10 ns;
```

Здесь 10 нс составляют половину периода тактового сигнала `clock`.

Написание теста

Первым шагом при написании тестового окружения является создание модуля VHDL, который выступает в качестве контейнера. Данный контейнер не имеет входных и выходных данных (см. рис. 1.11), поэтому при его описании входные и выходные порты отсутствуют, как показано в следующем фрагменте кода.

```
entity test_bench is
end test_bench;

architecture test of example_tb is
...
end architecture example_tb;
```

Архитектура тестового окружения должна содержать экземпляр тестируемого проекта (англ. *design under test – DUT*). Для создания экземпляра необходимо использовать те же языковые конструкции, что и при иерархическом описании проекта. Это означает, что можно создать экземпляр тестируемого модуля, используя либо оператор `component`, либо синтаксис непосредственного создания экземпляра. Таким образом, вторым шагом в написании теста является создание экземпляра тестируемого модуля. Для описанного ранее нами модуля `device_01` (см. листинг 1.3) этот шаг будет выглядеть следующим образом

```
DUT: entity work.device_01(Beh4) port map (x=>test_x, y=>test_y);
```

На заключительном шаге создания тестового окружения описывается генерация тестовых сигналов. В нашем случае, чтобы протестировать схему, нужно сгенерировать все восемь возможных двоичных комбинаций. Для этого мы присваиваем входным данным значение, а затем используем оператор `wait`, чтобы обеспечить задержку на заданный интервал времени. Следующий фрагмент кода иллюстрирует изложенные идеи.

```
stimulus: process
begin
    wait for 100ns;
    test_x <= "000";
    wait for 10ns;
    test_x <= "001";
    wait for 10ns;
    test_x <= "010";
    wait for 10ns;
    test_x <= "011";
    wait for 10ns;
    test_x <= "100";
    wait for 10ns;
    test_x <= "101";
    wait for 10ns;
    test_x <= "110";
    wait for 10ns;
    test_x <= "111";
    wait for 10ns;
end process stimulus;
```

Целиком VHDL-файл тестового окружения проекта приведен в листинге 1.4.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity device_01_tb is
end device_01_tb;

architecture Behavioral of device_01_tb is
signal test_x: std_logic_vector(2 downto 0);
signal test_y: std_logic;

begin

DUT: entity work.device_01(Beh4) port map (x=>test_x, y=>test_y);

stimulus: process
begin
```

```
    wait for 100ns;
    test_x <= "000";
    wait for 10ns;
    test_x <= "001";
    wait for 10ns;
    test_x <= "010";
    wait for 10ns;
    test_x <= "011";
    wait for 10ns;
    test_x <= "100";
    wait for 10ns;
    test_x <= "101";
    wait for 10ns;
    test_x <= "110";
    wait for 10ns;
end process stimulus;
```

Листинг 1.4. VHDL-описание тестового окружения

1.1.6. Маршрут моделирования цифрового устройства в САПР Vivado

В данном подразделе будет кратко рассмотрен общий маршрут моделирования цифрового устройства в САПР Vivado. Будут рассмотрены этапы разработки VHDL-описания устройства, создания тестового окружения, а также получения временных диаграмм работы устройства.

Чтобы создать проект в Vivado, после запуска необходимо выполнить в меню команду `File -> Project -> New`, как показано на рис. 1.12.

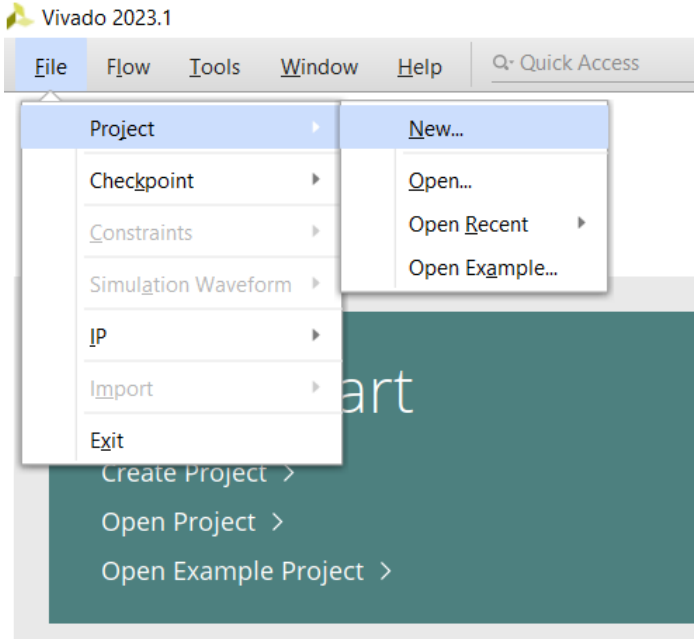

Рис. 1.12. Запуск процесса создания проекта в Vivado

После этого на экране появится окно мастера (*wizard*) для создания проекта в Vivado (рис. 1.13). Чтобы продолжить процесс, необходимо нажать `Next`.

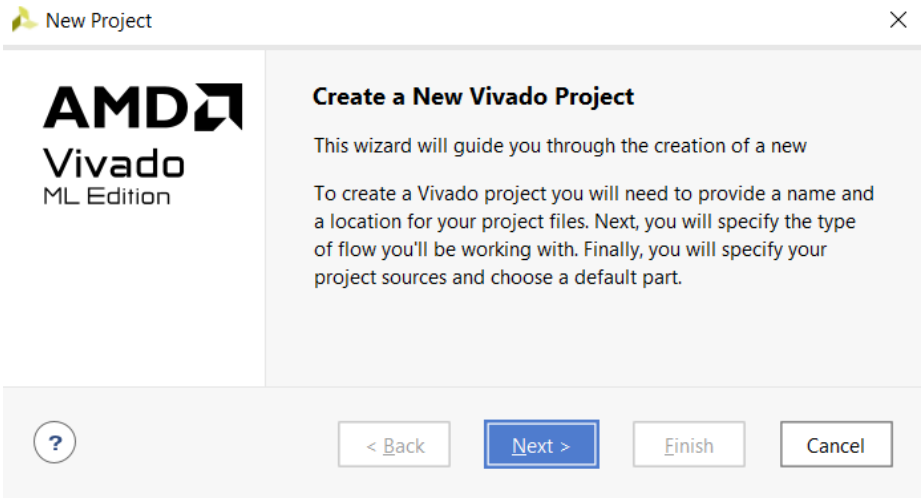

Рис. 1.13. Приветствие мастера по созданию проекта в Vivado

В появившемся окне мастера (рис. 1.14) необходимо ввести имя проекта (в данном случае это `lab_01`), а также при необходимости задать директорию на диске для хранения проекта. По завершении данных действий необходимо нажать `Next`.

В следующем окне мастера (рис. 1.15) будет предложено выбрать тип проекта: *RTL*-проект либо *Post-synthesis* проект. В большинстве случаев проекты описываются на уровне регистровых передач (*register transfer level – RTL*). Создаваемое в рамках лабораторной работы описание также относится к RTL-проекту. Если установить флаг `Do not specify sources at this time`, то Vivado пропустит этап добавления исходных файлов к проекту.

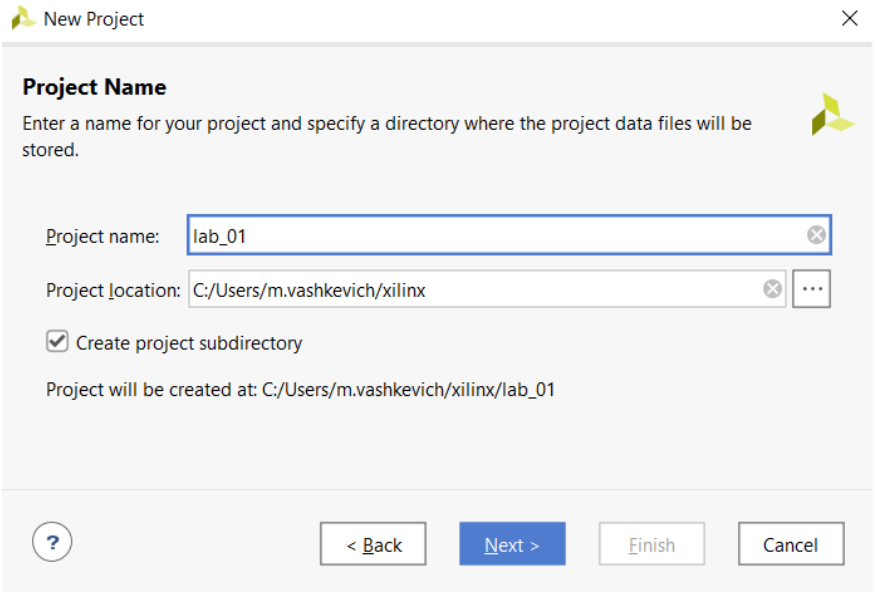

Рис. 1.14. Задание имени и папки проекта

Опция *Post-synthesis* проекта относится к тому случаю, когда исходное HDL-описание получено в результате работы сторонней программы-синтезатора, а главная цель проекта состоит в анализе аппаратных затрат ПЛИС, необходимых для реализации проекта.

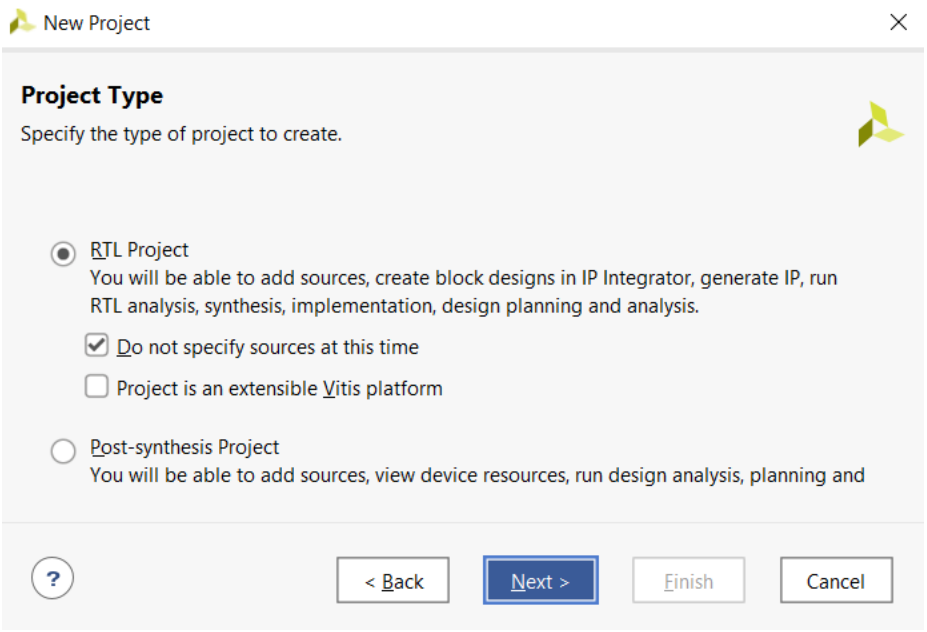

Рис. 1.15. Выбор типа проекта

Далее мастер создания проектов предложит выбрать конкретную микросхему ПЛИС либо целевую отладочную плату, для которой создается проект (рис. 1.16). В рассматриваемом случае никаких требований к аппаратной платформе нет, поэтому можно нажать `Next` и перейти к следующем этапу.

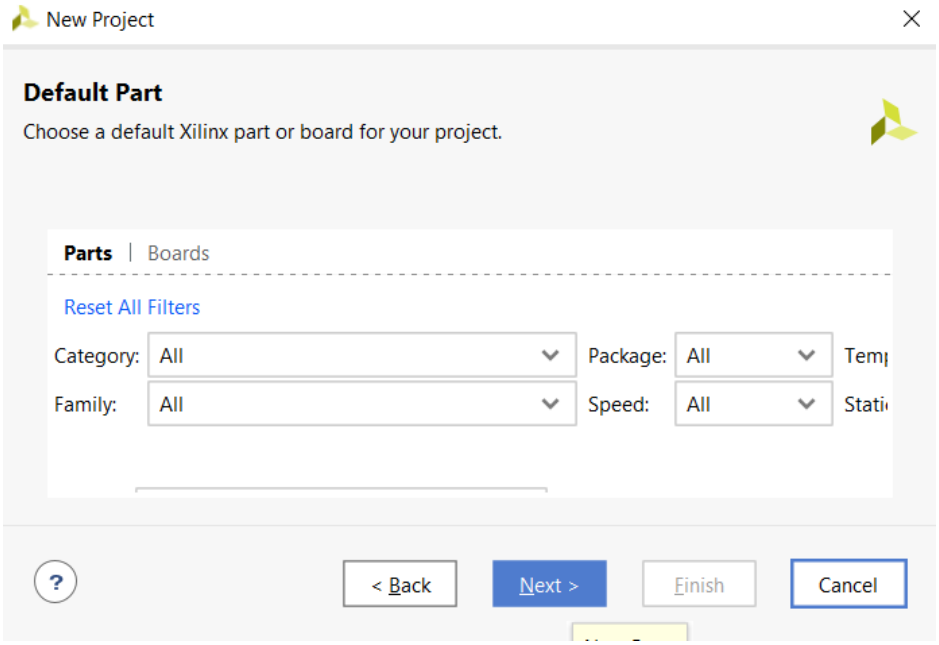

Рис. 1.16. Выбор аппаратной платформы

На заключительном этапе будет показано окно-уведомление (рис. 1.17) об окончании процесса создания проекта. В окне отображается название проекта, его тип, а также параметры аппаратной платформы, выбранные по умолчанию.

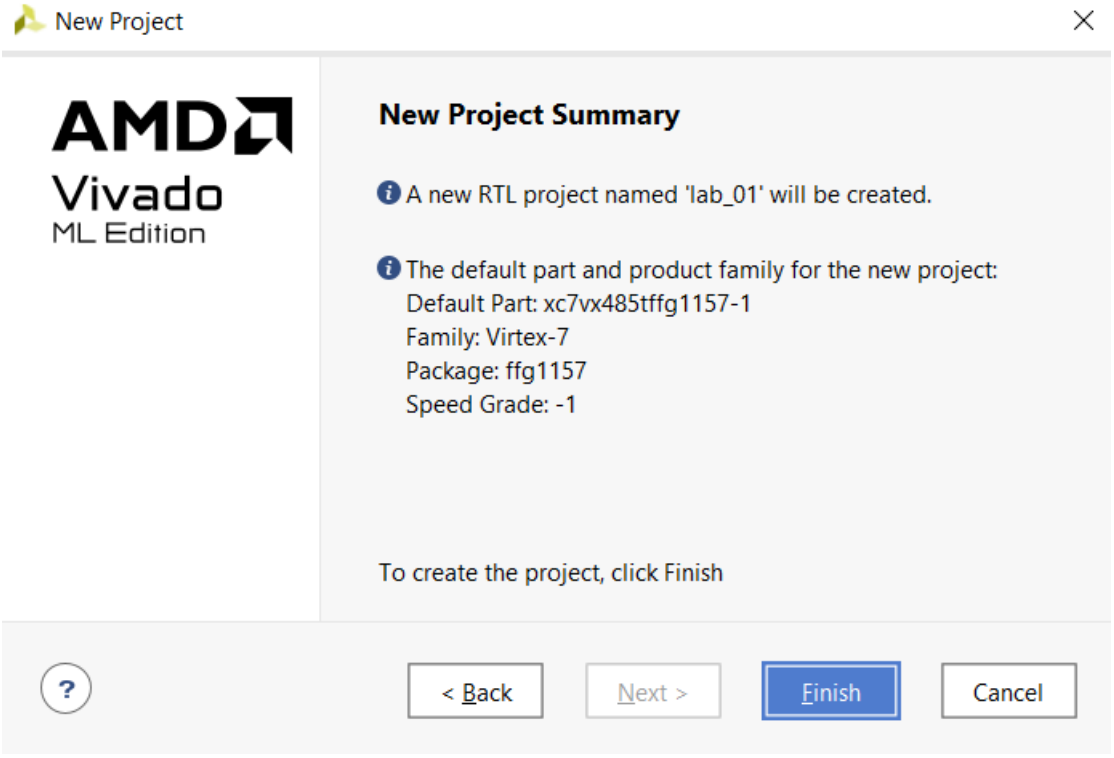

Рис. 1.17. Окно-уведомление об окончании процесса создания проекта

После окончания работы мастера откроется окно Vivado с проектом. Следующим этапом в создании проекта является добавление исходных файлов. Для этой цели в окне менеджера проекта необходимо выбрать пункт `Add Sources` (рис. 1.18).

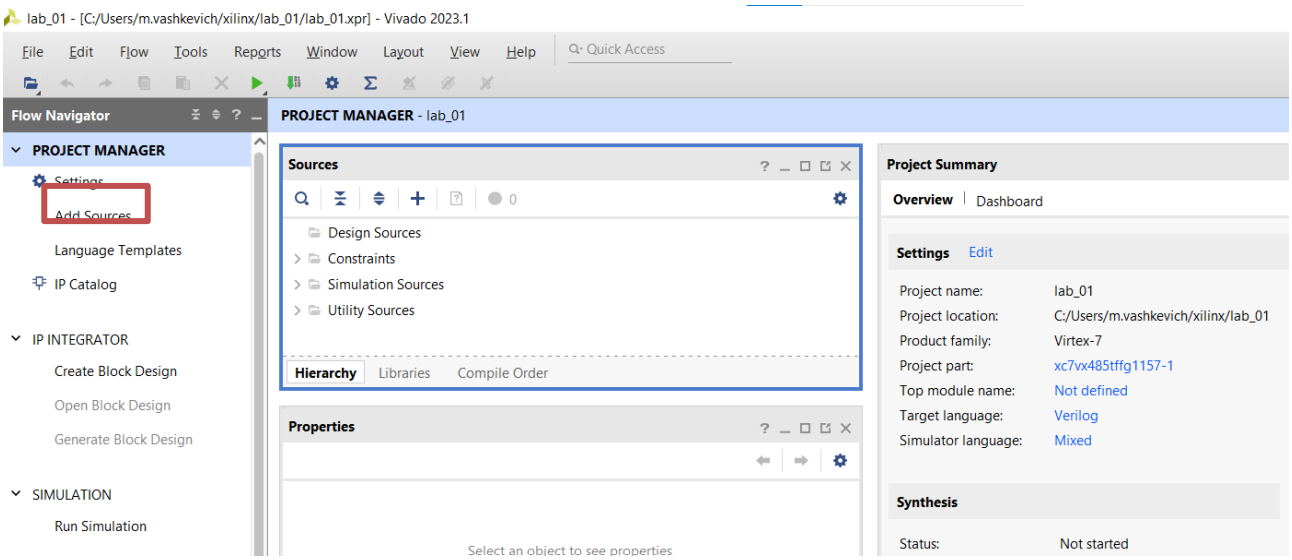

Рис. 1.18. Окно проекта Vivado

При этом на экране появится мастер добавления исходных файлов (рис. 1.19), в котором пользователю предлагается выбрать тип добавляемого исходного файла. Это может быть файл с ограничениями, исходный файл с описанием проекта на языках HDL либо файл, содержащий тест. С последними двумя типами мы уже познакомились выше. Файл с ограничениями (*constraints*) имеет важное значение на этапе создания загрузочного файла для FPGA. В нем, помимо прочего, описывается соответствие между портами разработанного устройства и выводами FPGA. На начальном этапе добавлять файл с ограничениями к проекту не нужно.

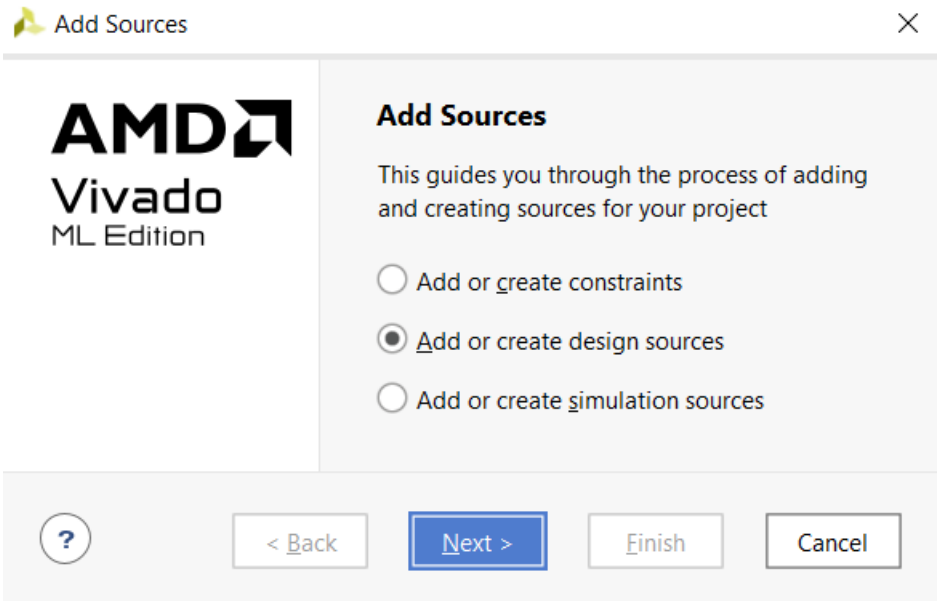

Рис. 1.19. Окно добавления исходных файлов

Для выполнения начального этапа разработки цифрового устройства необходимо добавить исходные файлы проекта (*design sources*), которые будут описывать все его блоки. Для этого нужно выбрать пункт `Add or create design sources` в окне добавления исходных файлов (см. рис. 1.19). После этого появится окно (рис. 1.20), в котором будет предложено добавить или создать исходные файлы проекта. Если необходимо добавить уже существующие файлы, то для этого нужно нажать кнопку `Add Files` и указать их место расположения. Если файлы необходимо создать (как в рассматриваемом случае), нужно нажать кнопку `Create File`. Далее появится окошко `Create Source File`, в котором будет предложено выбрать тип файла (VHDL, Verilog, SystemVerilog), имя файла и каталог для хранения файла. В данном случае выбираем тип файла VHDL, а имя – `AND2` (рис. 1.21). Аналогичным образом необходимо создать VHDL-файлы с именами `OR2` и `device_01`. По окончании этого действия окно должно выглядеть, как показано на рис. 1.22.

Далее мастер добавления исходных файлов предложит определить при помощи графического интерфейса входные и выходные порты модулей, а также их разрядность (рис. 1.23). Данный шаг можно пропустить, поскольку порты удобнее определять непосредственно в VHDL-коде.

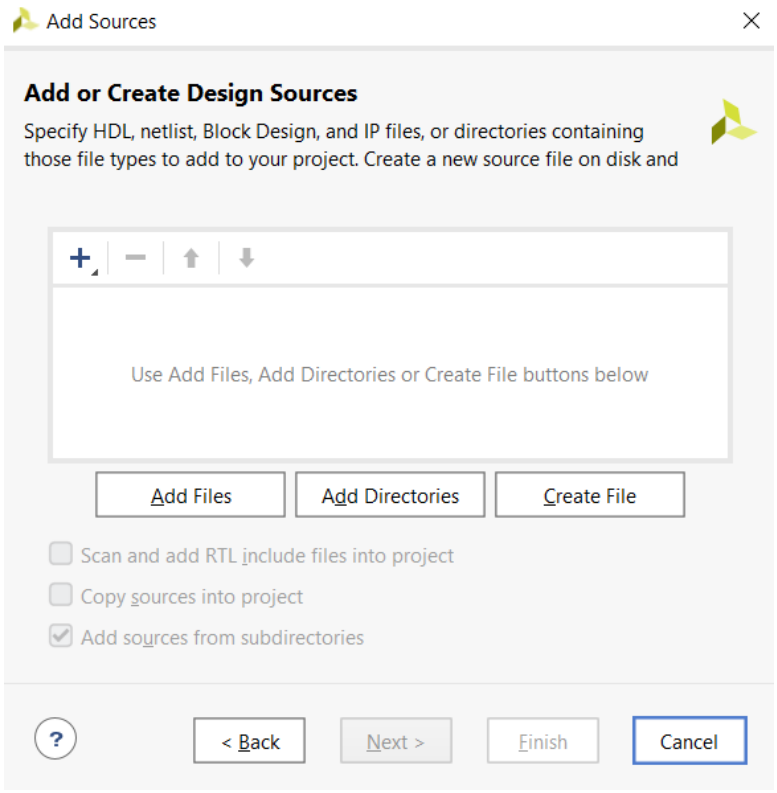

Рис. 1.20. Окно добавления/создания исходных файлов проекта

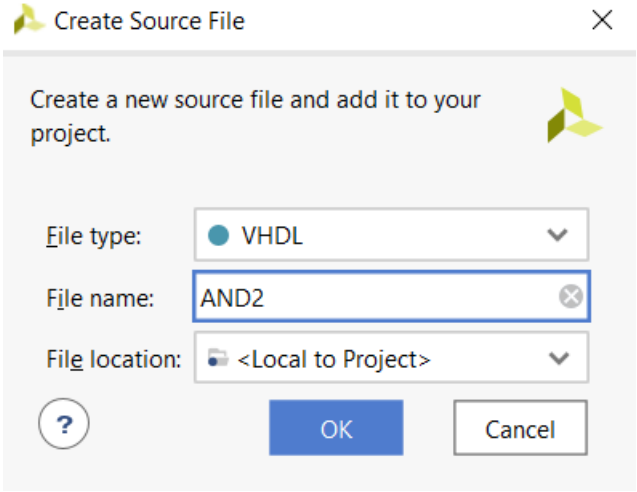

Рис. 1.21. Окно создания файла с описанием модуля `AND2`

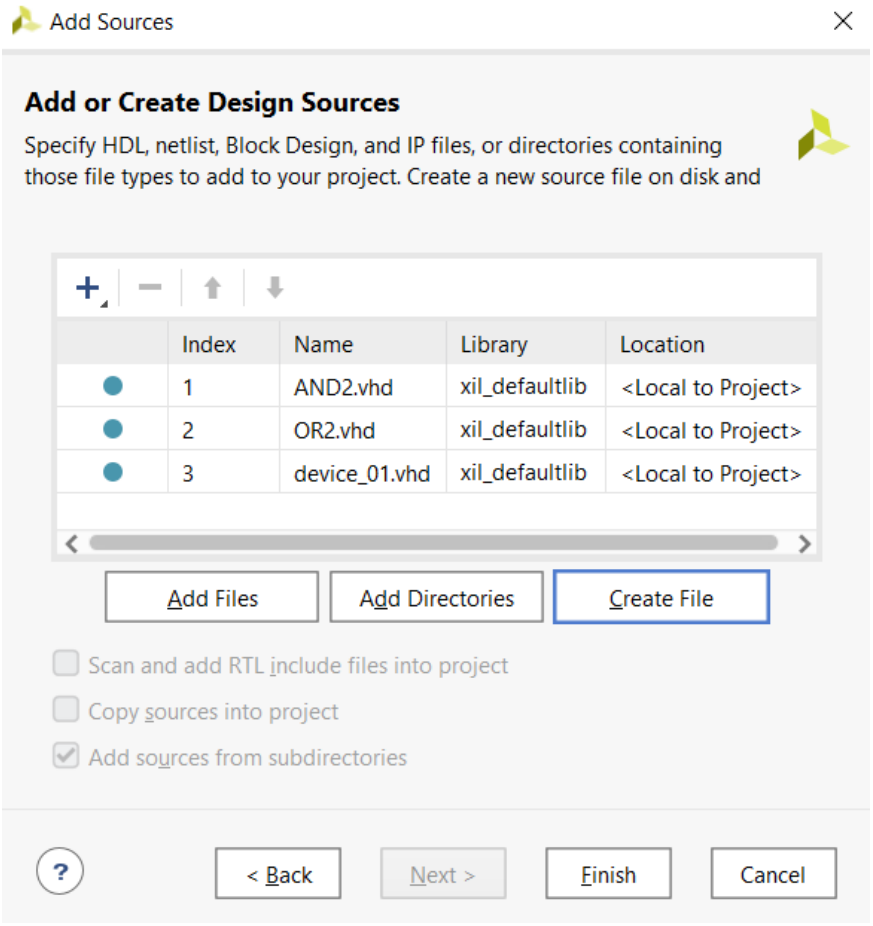

Рис. 1.22. Вид окна проекта после добавления исходных файлов проекта

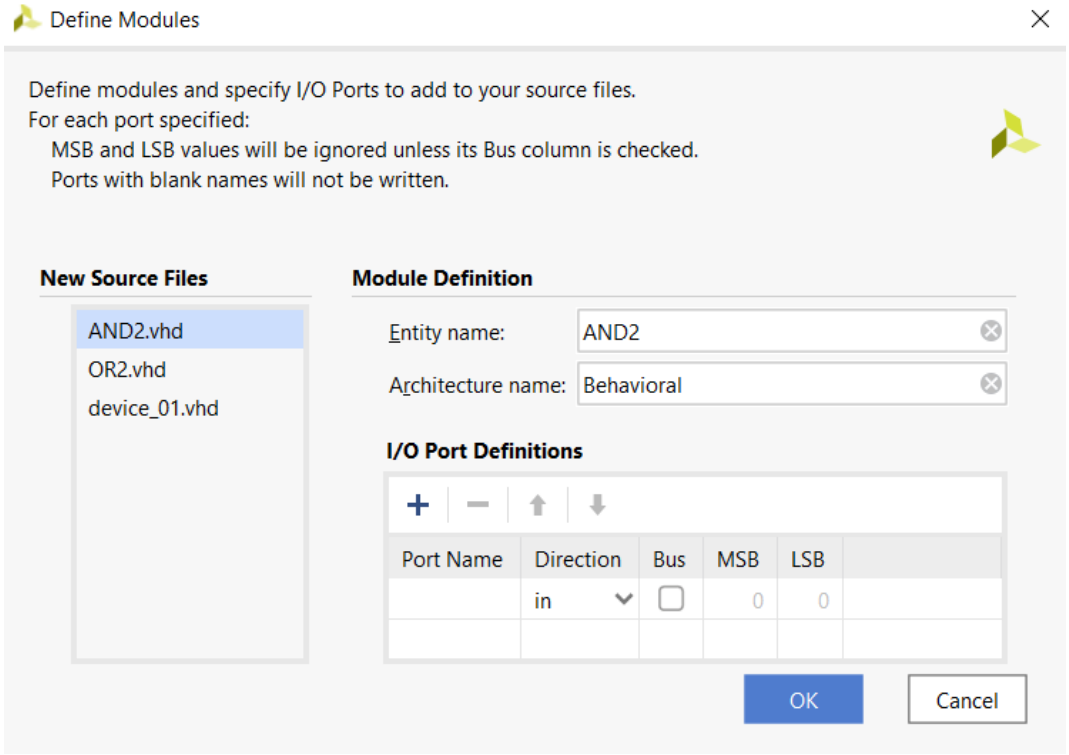

Рис. 1.23. Окно для задания портов модуля

После выполнения указанных действий в окне исходных файлов менеджера проектов должны отобразиться три новых исходных файла (рис. 1.24).

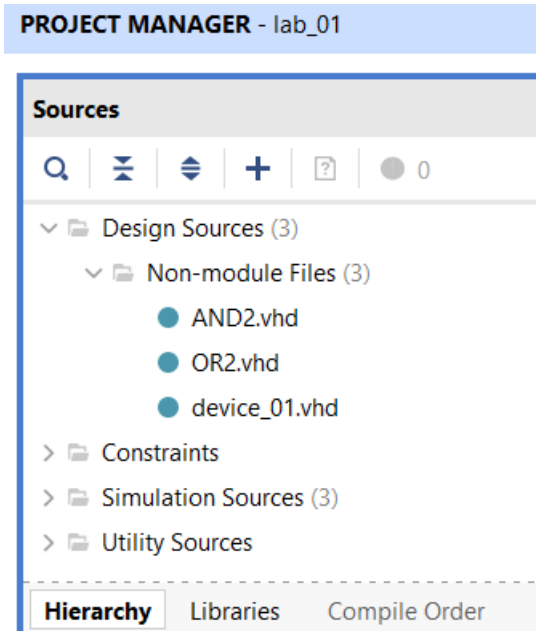

Рис. 1.24. Окно менеджера проекта

В проектные файлы `AND2.vhd` и `OR2.vhd` необходимо поместить описание вентилей, приведенное в листинге 1.2. В проектный файл `device_01.vhd` нужно поместить описание, приведенное в листинге 1.3. На этом завершается процесс описания схемы устройства.

Следующим этапом является тестирование проекта. Для этого необходимо к проекту добавить `vhdl`-файл с описанием теста. Для этого в окне менеджера проекта выбираем пункт `Add Sources` (см. рис. 1.18). Далее в появившемся окне (см. рис. 1.19) выбираем пункт `Add or create simulation sources` (создать или добавить файл для симуляции). В появившемся окне (рис. 1.25) нажимаем кнопку `Create File`, задаем имя файла (например, `device_test.vhd`), после чего нажимаем кнопку `Finish` для завершения процесса добавления тестовых файлов проекта. Vivado предложит определить входные и выходные порты создаваемого модуля (см. рис. 1.23), однако модуль, описывающий тест, не имеет входных и выходных портов, поэтому это действие можно пропустить. После окончания указанных действий в менеджере проектов в окне `Sources` в разделе `Simulation Sources` должен появиться файл `device_test.vhd` (рис. 1.26).

Для завершения процесса создания тестового окружения необходимо в файл `device_test` поместить описание теста, приведенное в листинге 1.4.

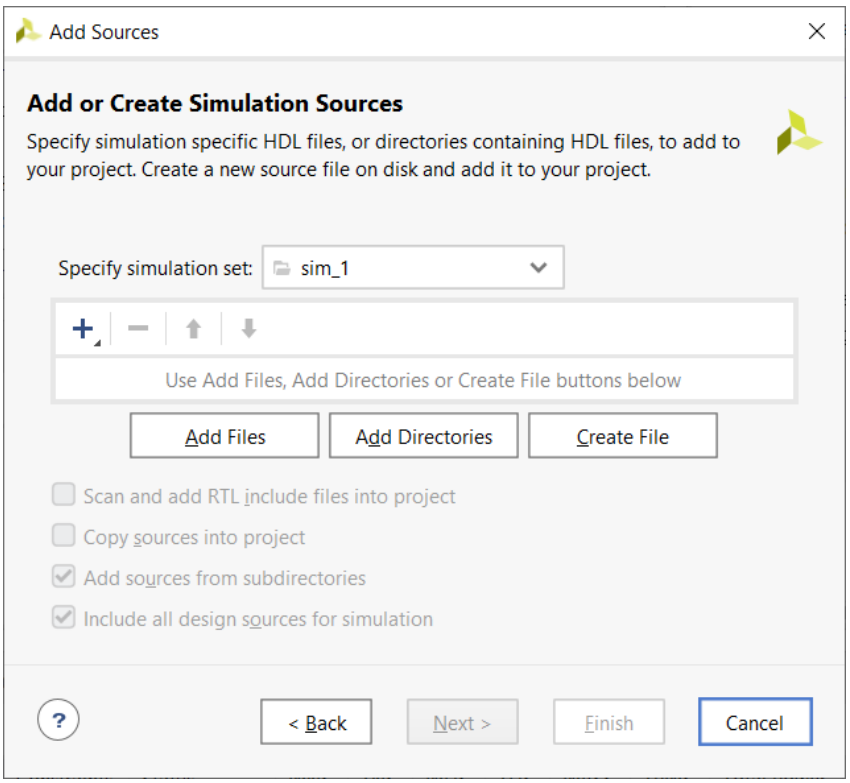

Рис. 1.25. Окно создания исходного файла с тестом

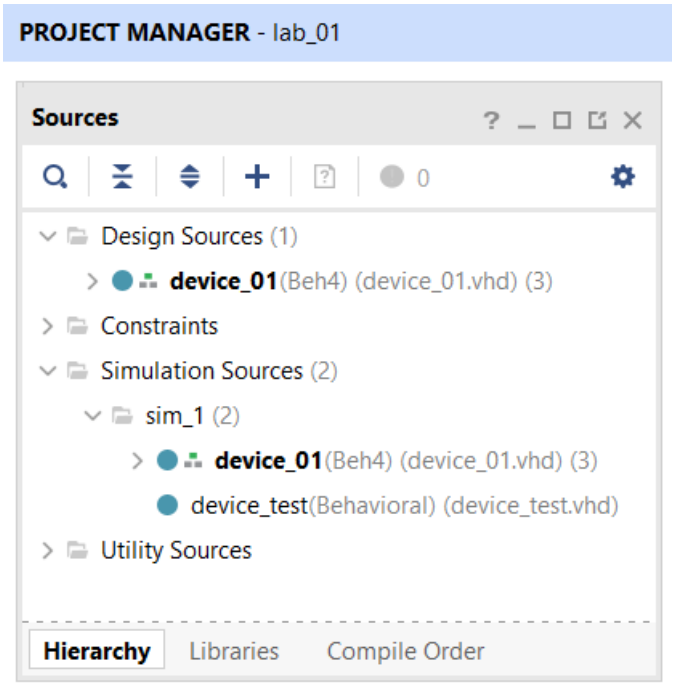

Рис. 1.26. Исходный файл с тестом (`device_test.vhd`)

Далее для запуска процесса моделирования необходимо на панели `Flow Navigator` выбрать пункт `Run Simulation`, как показано на рис. 1.27. После этого нужно выбрать первый появившийся пункт `Run Behavioral Simulation`.

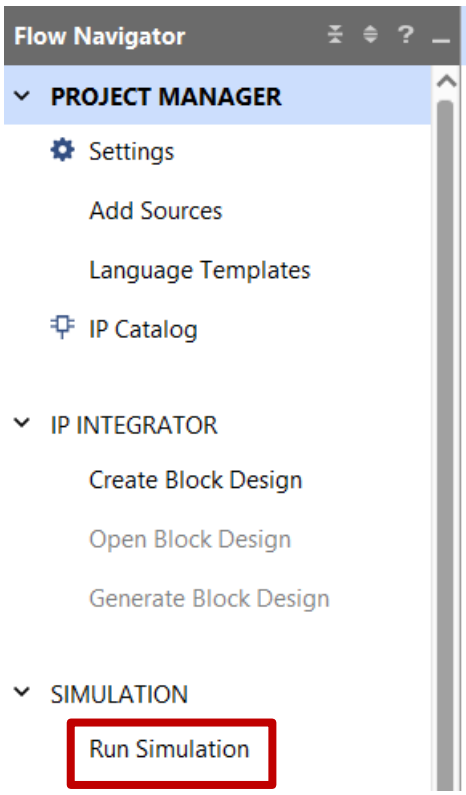

Рис. 1.27. Запуск процесса моделирования

В результате на экране появится окно с временной диаграммой теста, как показано на рис. 1.28.

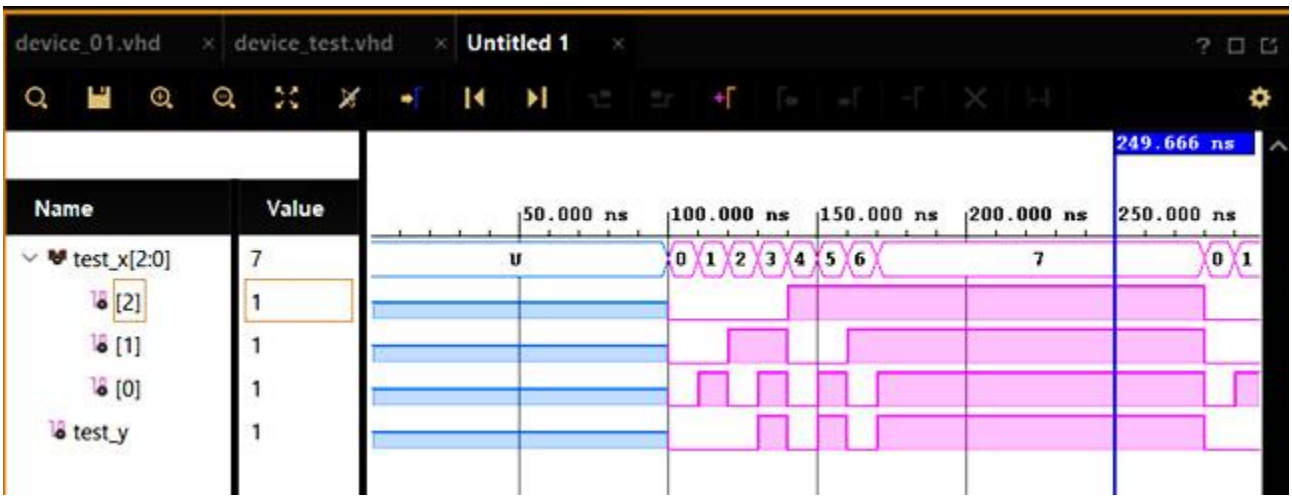

Рис. 1.28. Временная диаграмма теста

Вверху окна с временной диаграммой расположена панель инструментов и навигации, при помощи которой можно выполнить настройку внешнего вида временной диаграммы.

Таким образом, были рассмотрены этапы от создания исходного описания цифрового устройства до разработки теста и получения временных диаграмм работы устройства в САПР Vivado.

Помимо указанных возможностей, САПР Vivado позволяет анализировать логическую корректность вашего проекта. Чтобы убедиться, что составленное VHDL-описание компилируется без проблем, что все необходимые модели присутствуют и нет несоответствий интерфейсов, необходимо открыть окно `Shematic` (рис. 1.29) разработанного проекта (`RTL Analysis` → `Open Elaborated Design`)

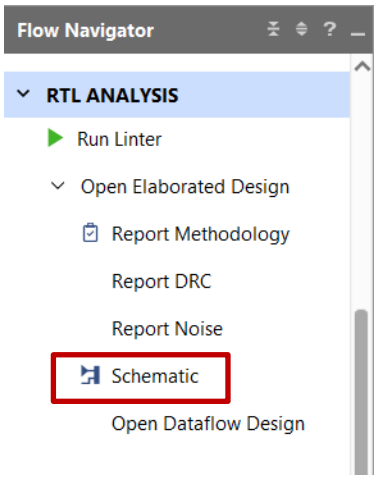

Рис. 1.29. Запуск процесса генерации схемы проекта

Например, для разработанного проекта (см. листинг 1.3) вид синтезированной схемы показан на рис. 1.30.

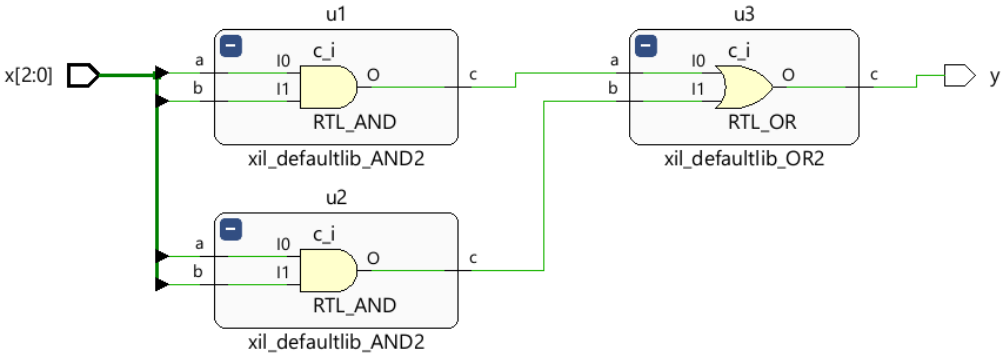

Рис. 1.30. Результат процесса анализа RTL-описания

В данном случае, поскольку анализируемое VHDL-описание не содержит сложных конструкций, все описанные примитивы были без труда распознаны Vivado как логические вентили. Полученная схема полностью совпадает с исходной (см. рис. 1.10).

Инструмент `Shematic` позволяет визуализировать логические взаимосвязи разработанных модулей, а также иерархическую структуру проекта. Важно отме-

тить, что в процессе анализа VHDL-описания не происходит отображения разработанного проекта на технологический базис ПЛИС. Данный процесс следует рассматривать как этап валидации разработанного VHDL-описания. Например, если согласно логике описания в схеме должен быть мультиплексор, а фактически в полученной схеме (см. рис. 1.30) его нет.

1.2. Порядок выполнения работы

1. Согласно варианту (табл. 1.2) выполнить синтез комбинационной схемы (рис. 1.31), реализующей систему логических функций. Для получения компактного описания логические функции необходимо минимизировать, применив любой известный метод минимизации, например, с помощью карт Карно, диаграмм двоичного выбора и т. д.

Таблица 1.2

Варианты для лабораторной работы № 1

Вариант	Система логических функций
1	$y_0 = (7, 10, 11, 13, 14, 15);$ $y_1 = (5, 6, 9, 12);$ $y_2 = (1, 4, 6, 9, 11, 12, 14)$
2	$y_0 = (3, 7, 8, 10, 11, 13, 14, 15);$ $y_1 = (2, 5, 6, 8, 9, 12, 15);$ $y_2 = (1, 3, 9, 11, 12, 13, 14, 15)$
3	$y_0 = (1, 3, 7, 11, 13, 15);$ $y_1 = (2, 3, 5, 6, 9, 12, 15);$ $y_2 = (1, 3, 6, 9, 11, 14)$
4	$y_0 = (0, 1, 2, 4, 6, 10, 11, 13, 14);$ $y_1 = (5);$ $y_2 = (2, 4, 5, 6, 8, 9, 10, 11, 12, 14)$
5	$y_0 = (3, 7, 8, 9, 10, 11, 12, 14, 15);$ $y_1 = (2, 3, 5, 6, 8, 9, 12, 15);$ $y_2 = (3, 6, 9, 11, 12, 14)$
6	$y_0 = (7, 13, 14);$ $y_1 = (5, 6, 8, 9, 12, 15);$ $y_2 = (1, 6, 9, 11, 12, 14)$
7	$y_0 = (2, 6, 7, 8, 9, 10, 11, 13, 15);$ $y_1 = (5, 9);$ $y_2 = (3, 11, 14)$
8	$y_0 = (7, 10, 11, 13, 14, 15);$ $y_1 = (2, 3, 5, 6, 8, 9, 12, 15);$ $y_2 = (0, 1, 3, 5, \dots, 15)$

2. Разработать структурное VHDL-описание разработанной схемы.

3. Разработать поведенческую VHDL-модель разработанной схемы.

4. Составить тестирующую программу, выполнить моделирование разработанных VHDL-описаний на всех наборах значений входных переменных.

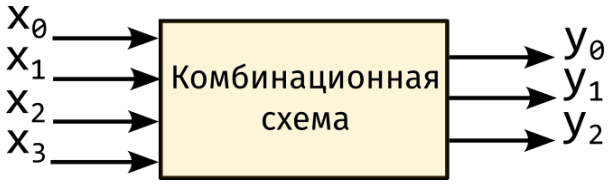

Рис. 1.31. Комбинационная схема, реализующая систему булевых функций

5. Составить отчет. В отчете должны быть приведены таблица истинности и соответствующие ей VHDL-модели. Также в отчете должна содержаться тестирующая программа для всех наборов входных переменных, соответствующих таблице истинности. В обязательном порядке в отчете должны быть приведены временные диаграммы, соответствующие тестирующей программе.

6. Подготовить отчет по лабораторной работе.

1.3. Дополнительные вопросы и задания

1. Записать семь логических операций языка VHDL.

2. Можно ли записывать оператор назначения сигнала как в левую (`<=`), так и в правую (`=>`) сторону?

3. Имеет ли тестирующая программа входные и выходные порты?

4. Используя логические операторы, записать VHDL-модель многоуровневой комбинационной схемы.

5. Верно ли то, что архитектурное тело (`architecture`) есть множество параллельных операторов, взаимодействующих между собой и находящихся под влиянием друг друга?

ЛАБОРАТОРНАЯ РАБОТА № 2. ВРЕМЕННОЕ МОДЕЛИРОВАНИЕ ЦИФРОВЫХ УСТРОЙСТВ

ЦЕЛЬ РАБОТЫ: создание VHDL-модели электронной схемы, моделирование ее в САПР Vivado с задержками и без, анализ полученных временных диаграмм и определение максимальной задержки схемы.

2.1. Теоретические сведения

2.1.1. Задержка распространения сигнала

Все физические схемы имеют задержку распространения сигнала между изменением входного сигнала и результирующим изменением выходного сигнала. В качестве примера рассмотрим функционирование обычного инвертора (рис. 2.1).

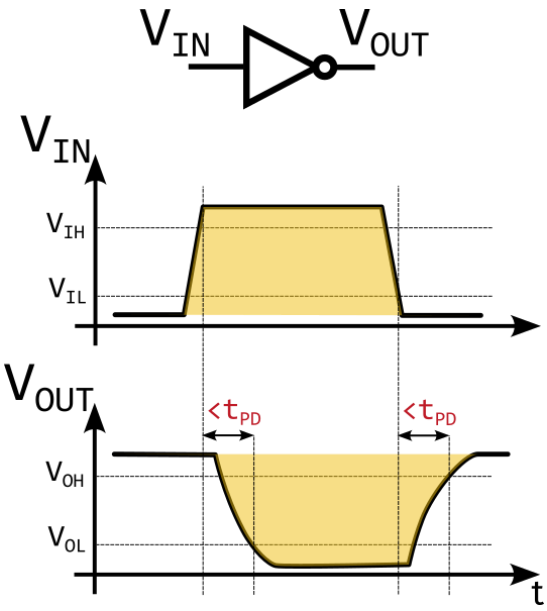

Рис. 2.1. Задержка распространения сигнала в инверторе

Когда входное напряжение V_{IN} достигает уровня V_{IH} – логической единицы (по входу) – начинается переходной процесс переключения выходного уровня инвертора из состояния логической единицы в состояние логического нуля. Процесс считается завершенным, когда напряжение на выходе инвертора опускается ниже V_{OL} (выходного уровня нуля). Задержка между входом и выходом обозначается как t_{PD} и именуется задержкой распространения сигнала (*propagation delay*). В данном случае t_{PD} – это максимальная задержка распространения, которая в документации обычно обозначается как $t_{PD,MAX}$. Аналогичные процессы происходят и при переключении из состояния логической единицы в состояние логического нуля. Важно отметить, что время задержки при переключении из нуля в единицу и из единицы в нуль могут отличаться. Разумеется, при проектировании цифровых устройств стремятся к тому, чтобы минимизировать задержку распространения сигнала.

Помимо максимальной задержки распространения сигнала (см. рис. 2.1), иногда рассматривают также минимальную задержку распространения $t_{PD,MIN}$. В англоязычных источниках ее обозначают как t_{CD} (англ. *contamination delay*). По сути, t_{CD} представляет из себя нижнюю границу задержки от момента, когда сигнал перестал быть валидным на входе, до момента, когда сигнал перестает быть валидным на выходе, как показано на рис. 2.2.

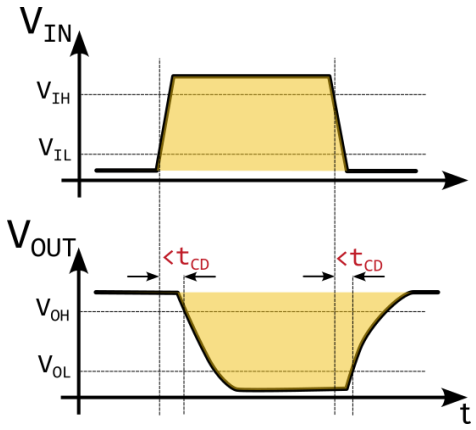

Рис. 2.2. Минимальная задержка распространения сигнала

Если время t_{CD} не задано, его можно считать равным нулю. Таким образом, временную диаграмму переключения инвертора можно изобразить так, как показано на рис. 2.3.

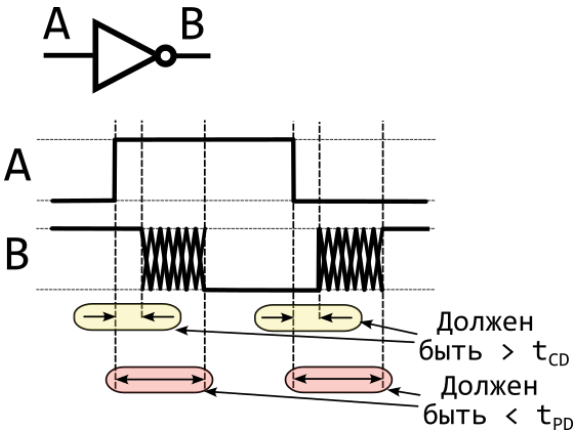

Рис. 2.3. Диграмма переключения инвертора

Во время переходного процесса, пока от момента переключения входного сигнала `A` прошло менее t_{PD}, состояние выхода `B` считается неопределенным. Значение выхода в этот период времени может восприниматься другими схемами как логическая единица либо как логический нуль. После истечения времени задержки распространения сигнала выход элемента переходит в устойчивое состояние.

Наличие у реальных логических элементов задержек распространения приводит к тому, что на выходах цифровых схем, собранных из логических элементов,

могут возникать паразитные импульсы (англ. *glitch*), которые также называют рисками сбоя или просто рисками (англ. *hazard*). В качестве иллюстрирующего примера рассмотрим схему на рис. 2.4.

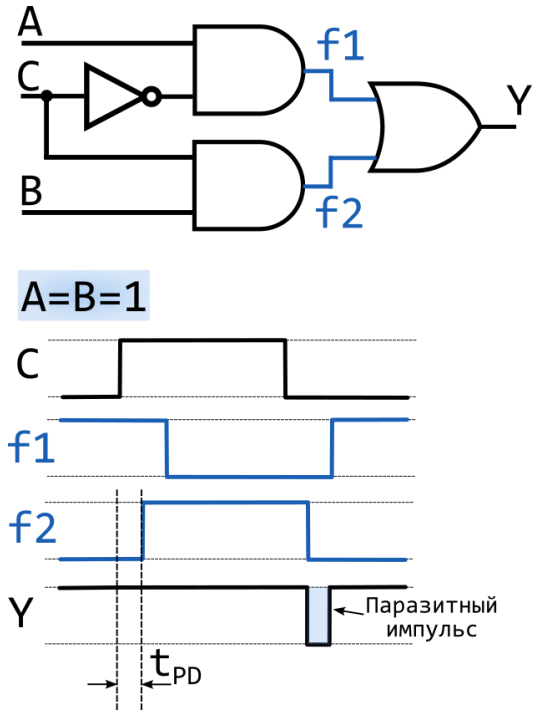

Рис. 2.4. Схема, иллюстрирующая механизм появления паразитного импульса на выходе цифрового устройства

В рассматриваемой схеме входы `A` и `B` считаются фиксированными, на них подана логическая единица. Изменяется только сигнал `C`, который в данной ситуации не должен влиять на выходное значение `Y`. Тем не менее существование у реальных логических элементов задержек приводит к тому, что внутренние сигналы `f1` и `f2` изменяются не синхронно. В частности, сигнал `f1` отстает от `f2`, поскольку для его изменения должно произойти переключение двух элементов, а изменение `f2` зависит только от одного вентиля И. Разбежка в переключении внутренних сигналов приводит к тому, что возникает ситуация, при которой выход `Y` на короткое время переходит в состояние логического нуля. Данный пример является иллюстрацией статического риска (сбоя). Вообще, под статическим риском понимают кратковременное изменение сигнала, который должен был бы оставаться неизменным (единичным или нулевым). Если же состояние выхода должно изменяться, но вместо однократного перехода происходят многократные, то имеет место динамический риск.

2.1.2. Анализ временных задержек

Важным элементом процесса проектирования цифрового устройства является анализ временных задержек разработанной схемы. В процессе анализа временных задержек все узлы в схеме помечаются как входные, выходные либо внутренние. Средство анализа (англ. *timing analyzer*) вычисляет время прибытия (англ. *arrival time*) для всех узлов схемы (т. е. максимальное время, за которое каждый из узлов схемы переключится). При этом разработчик должен указать время поступления входных сигналов, а также время, когда данные должны появиться на входе. Время переключения a_i узла i зависит от задержки распространения вентиля, который управляет (является драйвером) узлом i, а также от времени переключения входов этого вентиля.

$$a_i = \max_{j \in fan_{in}(i)} \{a_j\} + t_{PD_i}.$$

При анализе задержек важным понятием является *временной резерв* (англ. *slack*), который определяется как время между требуемым и фактическим временем переключения сигнала. Если временной резерв является положительным, то считается, что цифровая схема удовлетворяет временным ограничениям. Отрицательное значение временного резерва указывает на то, что схема работает недостаточно быстро.

Чтобы глубже разобраться с понятием временного резерва, рассмотрим схему на рис. 2.5. На этой схеме внутри вентилей указаны их задержки в пикосекундах.

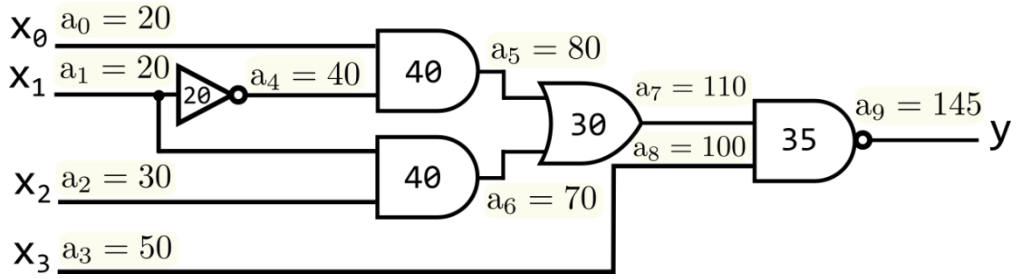

Рис. 2.5. Схема, иллюстрирующая понятие временного резерва (*slack*)

Для каждого узла указано время поступления сигнала. Для внутренних узлов это время поступления рассчитывается с использованием выражения приведенной формулы. Если дополнительно задать, что ожидаемое время поступления сигнала на выход равно 200 пс, то временной резерв в данном случае будет равен 55 пс.

Как правило, большинство цепей в цифровом устройстве удовлетворяет заданным временным ограничениям, при этом, однако, может существовать некоторое число критических путей, которые ограничивают скорость работы устройства. На рис. 2.6 показан критический путь для схемы, которая использовалась для иллюстрации понятия временного резерва.

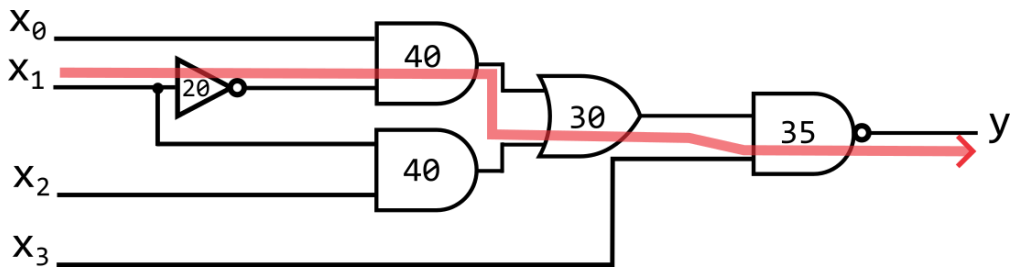

Рис. 2.6. Критический путь схемы

В рассматриваемом примере критический путь проходит через четыре вентиля. Общая задержка, которая возникает между приходами входных сигналов и формированием выхода, равна 125 пс.

2.1.3. Моделирование VHDL-описаний цифровых устройств

При моделировании VHDL-описаний схем иногда необходимо указать величину задержки между входом и выходом ее элементов. Ниже показан пример, который поясняет, каким образом можно добавить задержку в описание простого вентиля И (листинг 2.1).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND2 is
    Port (a: in STD_LOGIC;
          b: in STD_LOGIC;
          c: out STD_LOGIC);
end AND2;

architecture arch1 of AND2 is
begin
c <= a and b after 2 ns;
end arch1;
```

Листинг 2.1. VHDL-описание вентиля И с задержкой

Главной особенностью приведенного описания является использование ключевого слова `after` в операторе назначения сигнала `<=`. Фактически в архитектурном описании вентиля И указывается, что новое значение выходному сигналу `c`, равное `a and b`, будет назначено через 2 нс модельного времени после изменения входных сигналов (`a` либо `b`). Более формально оператор назначения сигнала имеет следующее описание.

```
<signal> <= [transport] <value> [after <time>];
```

Здесь, как и ранее, в квадратных скобках указаны необязательные параметры. Ключевое слово `transport` определяет, что при назначении сигнала используется *транспортная* задержка, в противоположном случае задержка считается *инерционной*.

В случае инерционной задержки значение сигнала изменяется только тогда, когда входной сигнал будет сохранять соответствующий уровень в течение времени, заданного параметром `after`. В случае транспортной задержки сигнал изменяется независимо от того, сколько входной сигнал сохранял свой уровень. Приведем пример.

```
y1 <= not x after 10ns;  -- назначение сигнала
y2 <= transport not x after 10ns;
```

На рис. 2.7 показана временная диаграмма сигналов `x`, `y1` и `y2`.

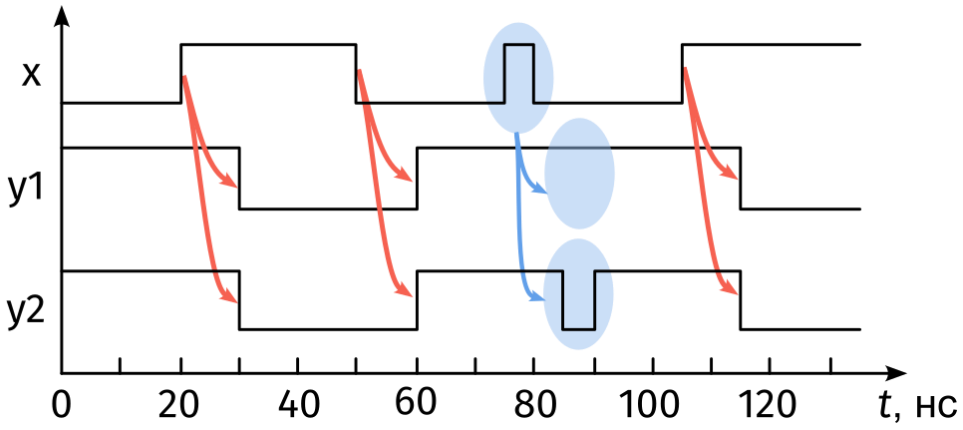

Рис. 2.7. Иллюстрация понятия транспортной и инерционной задержки

Можно заметить, что поведение `y1` и `y2` практически везде совпадает, за исключением временного интервала, в котором во входном сигнале `x` возникает кратковременный импульс длительностью менее 10 нс. В сигнале `y2` импульс находит свое отражение, поскольку в данном случае моделируется транспортная задержка сигнала, которая используется, чтобы моделировать задержки, возникающие в линиях связи между отдельными элементами схемы. В сигнале `y1` реакции на импульс нет, поскольку моделируемая задержка в данном случае является инерционной, поэтому любые изменения на входе длительностью менее 10 нс отфильтровываются, т. е. отбрасываются системой моделирования и не приводят к формированию нового значения `y1`.

Рассмотрим более подробно, как происходит процесс моделирования работы цифрового устройства, описанного на языке VHDL. Ранее уже упоминалось, что сигналы в цифровых устройствах передаются и обрабатываются параллельно. Обратимся к примеру, показанному на рис. 2.8.

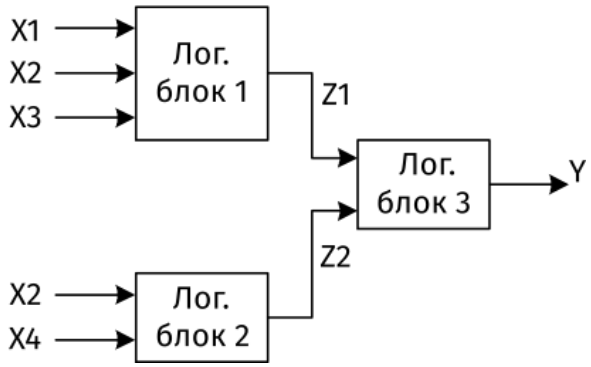

Рис. 2.8. Параллельные процессы в VHDL

Если подать значения входных сигналов на логический блок 1 и логический блок 2, то эти блоки будут одновременно активированы. Через некоторое время выходные сигналы поступят на вход блока 3, а в это время на блоки 1 и 2 могут прийти новые значения.

Параллельность работы отдельных блоков реализуется на VHDL с помощью процессов (`process`). Каждому логическому блоку соответствует свой процесс. Описание схемы может иметь следующий вид.

```
…
BL1: process (x1,x2,x3)
     begin
     end process BL1;
BL2: process (x2,x4)
     begin
     end process BL2;
BL3: process (z1,z2)
     begin
     end process BL3;
…
```

Процесс активируется, когда происходит изменение какого-либо сигнала в списке сигналов запуска (списке чувствительности) этого процесса, который идет после ключевого слова `process`. Список чувствительности соответствует входным сигналам логического блока.

Каждый процесс может быть в одном из трех состояний:

– выполняющийся;
– активный;
– приостановленный.

Процесс является выполняющимся, когда система моделирования выполняет процесс; активным, когда процесс ожидает, чтобы система его выполнила; приостановленным, когда не является активным или выполняющимся. Выполненный процесс является приостановленным.

Параллелизм процессов программно имитируется, поэтому только один из нескольких параллельных процессов является выполняющимся по существу. Моделирование параллельных процессов осуществляется так: все параллельные процессы, которые надо одновременно выполнить, выстраиваются в очередь, система моделирования выбирает процесс из очереди и выполняет его, затем следующий, пока не будет исчерпан весь список активных процессов. Когда очередь активных процессов пуста, считается, что все параллельные сигналы выполняются одновременно, и тогда изменяются значения всех сигналов. После этого выполняется следующий цикл моделирования.

В процессе моделирования одного дискретного момента времени приостановленный процесс может стать активным, когда один из драйверов сигналов в списке чувствительности изменился.

Чтобы пояснить процесс моделирования VHDL-описаний, рассмотрим еще один пример схемы (рис. 2.9), которая описывается VHDL-кодом, приведенным ниже.

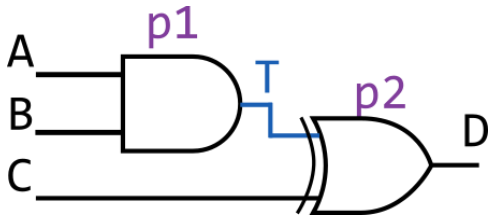

Рис. 2.9. Комбинационная схема

```
entity NANDXOR is
port (A,B,C: in std_logic; D: out std_logic);
end NANDXOR;

architecture RTL of NANDXOR is
signal T: std_logic;
begin
 p0: T<=A nand B after 2 ns;
 p1: process (T,C)
     begin
      D<=T xor C after 3 ns;
     end process p1;
end RTL;
```

При инициализации каждый из сигналов имеет значение 0. Если в ходе теста какой-то из входных сигналов изменился, то процесс `p0` вычислит значение `T` и приостановится, затем будет вычисляться значение `D` в процессе `p1`. Очередной цикл моделирования начинается тогда, когда один из сигналов изменится. Изменение сигнала называется событием. На рис. 2.10 показана временная диаграмма работы комбинационной схемы при изменении входных сигналов.

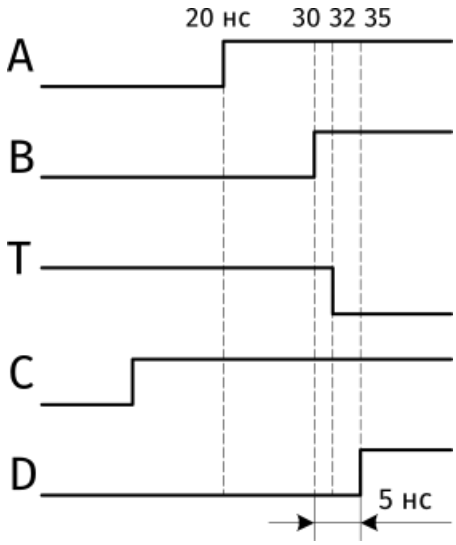

Рис. 2.10. Временная диаграмма работы комбинационной схемы

Из приведенной временной диаграммы видно, что максимально возможная задержка (соответствующая критическому пути схемы) равна 5 нс и возникает при переключении входного сигнала B.

2.1.4. Дельта-задержка

Для правильного понимания моделирования VHDL-описания важным является понятие *дельта-задержки*. Чтобы пояснить данное понятие, рассмотрим, как будет происходить процесс моделирования комбинационной схемы, показанной на рис. 2.9, если не учитывать задержек распространения сигналов в элементах схемы. В этом случае архитектурное тело VHDL-описания схемы примет следующий вид.

```
architecture Delta of NANDXOR is
signal T: std_logic;
begin
 p0: T<=A nand B; -- нет after
 p1: process (T,C)
     begin
      D<=T xor C;
     end process p1; -- нет after
end Delta;
```

Временная диаграмма, получаемая в результате моделирования данного описания, представлена на рис. 2.11.

В момент времени 30 нс входной сигнал B изменился, что послужило причиной изменения сигнала T и сигнала D. На временной диаграмме все это случилось в одно время, однако логичным будет сказать, что сигнал T изменился после сигнала B. Бесконечно малая величина, показанная на рис. 2.11, называется дельта-задержкой. Сигнал D изменится после изменения сигнала T, т. е. появляется другая дельта-задержка.

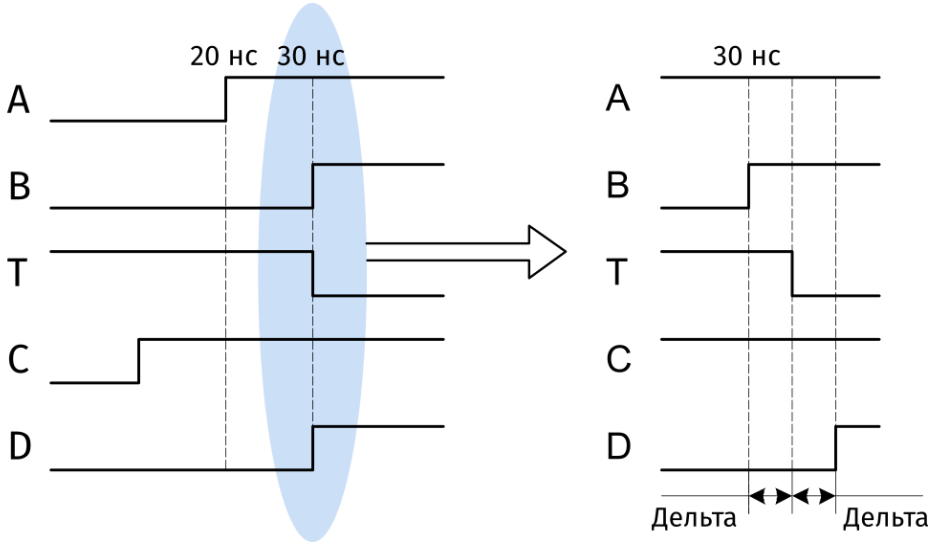

Рис. 2.11. Иллюстрация понятия дельта-задержки

Дельта-задержка – основное понятие моделирования в VHDL, это условная величина (один цикл прогона VHDL-модели, условный такт моделирования, когда нет явного указания времени задержки – `after`, `wait`).

VHDL-модели, имитирующие поведение цифровых систем, состоят из множества процессов. Во время прогона модели в данном цикле запускаются все процессы, входные сигналы которых изменились с момента выполнения последнего цикла. После того как все процессы выполнены, данный цикл моделирования считается завершенным. Следующий цикл начинается тогда, когда произойдет изменение какого-либо сигнала в списке чувствительности процесса. Этот период может составлять секунды, наносекунды или быть равным дельта. С точки зрения логической схемы дельта-задержка – задержка на одном каскаде логики, когда не заданы конкретные временные задержки логических элементов.

Механизм событий при моделировании позволяет имитировать исполнение параллельных процессов несмотря на то, что работа моделирующей программы представляет собой один поток операций, выполняющихся последовательно. Механизм дельта-задержек позволяет обнаружить некорректные присваивания сигналов типа `X <= not X`. В этом случае процесс моделирования заходит в бесконечный цикл. Чтобы этого избежать, задают ограничение на число тактов моделирования (обычно 5000) и прерывают моделирование, если за это количество тактов не произошло изменение ни одного сигнала за нулевой момент времени (за дельта-задержку).

Важным является вопрос о том, где в описаниях на языке VHDL могут быть использованы операторы присваивания значений переменным и операторы назначения сигналов. Присваивание значений переменным разрешается внутри процессов, в функциях и процедурах. Назначение сигнала может встречаться в любом месте исполняемого раздела архитектурного тела.

2.2. Порядок выполнения работы

1. Получить у преподавателя вариант нерегулярной логической схемы (см. приложение). Для заданной схемы составить структурное VHDL-описание. Составить VHDL-модель каждого из типов элементов, входящих в схему. Если в схеме есть элементы одинакового типа, то составляется одна модель для всех элементов данного типа. Модель элемента должна соответствовать задержке, указанной в таблице П.1. При графическом изображении логического элемента на схеме будут указываться его тип (библиотечное имя) и имена входных и выходных полюсов.

2. Составить тестирующую программу для всех наборов значений входных переменных.

3. Провести моделирование и получить временную диаграмму.

4. По временной диаграмме записать систему логических функций, реализуемых схемой.

5. Для каждого тестирующего набора определить задержку схемы.

6. Найти критический путь на схеме – путь с наибольшей суммарной задержкой элементов.

7. Подготовить отчет по лабораторной работе. В отчете должна быть нарисована логическая схема. При этом обозначения сигналов, элементов схемы должны соответствовать описанию на языке VHDL. В отчете должен содержаться VHDL-код схемы и тестирующая программа. VHDL-код и тест должны быть в отдельных файлах и содержать комментарии: а) автор разработанной VHDL-модели; б) номер варианта. В отчете должны содержаться временные диаграммы, соответствующие тестирующей программе, система логических функций, реализуемых схемой. На логической схеме должен быть отмечен критический путь. В отчете должно быть указано значение задержки схемы, соответствующее задержке критического пути.

2.3. Дополнительные вопросы и задания

1. Нарисовать дерево иерархии проекта логической схемы, моделирование которой было проведено лабораторной работе № 2.

2. Что такое ключевое соответствие портов?

3. Что такое позиционное соответствие портов?

4. Могут ли употребляться операторы создания экземпляров компонентов (`port map`) вместе с операторами назначения сигнала в архитектурном теле?

5. Правильно ли то, что в операторе `port map` символы `=>` и `<=` (соответствия) употребляются в зависимости от направления порта (для входа символы `=>`, для выхода символы `<=`)?

6. Правильно ли то, что операторы `port map` обязательно должны иметь метки?

7. Что такое временной резерв (*slack*)?

8. Что такое транспортная и инерционная задержки в VHDL? Для чего они используются?

9. Пояснить, что такое дельта-задержки и для чего они используются в процессе моделирования.

ЛАБОРАТОРНАЯ РАБОТА № 3. ПРОЕКТИРОВАНИЕ ЦИФРОВЫХ УСТРОЙСТВ НА УРОВНЕ ВЕНТИЛЕЙ И РЕГИСТРОВЫХ ПЕРЕДАЧ НА SYSTEMVERILOG

ЦЕЛЬ РАБОТЫ: изучение основных конструкций языка SystemVerilog, освоение концепции проектирования цифровых устройств на уровне логических вентилей и регистровых передач.

3.1. Теоретические сведения

3.1.1. Краткая история SystemVerilog

Язык описания аппаратуры Verilog создан в начале 1980-х годов компанией Gateway Design Automation. Вначале Verilog был проприетарным программным обеспечением (ПО), однако в 1989 году он перешел в категорию ПО с открытым исходным кодом. В 1995 году был выпущен первый стандарт IEEE данного языка. Стандарт впоследствии обновлялся в 2001 и 2005 годах. Версия 2001 года является наиболее широко используемой, при ссылке на нее обычно говорят просто Verilog. Однако растущая сложность проектирования цифровых схем привела к тому, что в 2005 году был выпущен новый IEEE стандарт Verilog. В это же время был разработан новый стандарт, описывающий язык SystemVerilog. После 2005 года в IEEE началась работа по объединению двух стандартов в один язык. Последующий язык получил название SystemVerilog и был закреплен в стандарте 2009 года. На этом этапе Verilog был полностью заменен SystemVerilog. Впоследствии происходило обновление стандарта SystemVerilog в 2012 и в 2017 годах. Некоторые расширенные функции SystemVerilog остаются нереализованными в различных коммерческих компиляторах SystemVerilog.

3.1.2. Модуль

В SystemVerilog *модуль* является базовым «строительным блоком». Каждый модуль определяется как набор входных и выходных *портов*, которые являются входными и выходными сигналами схемы. Тело каждого модуля начинается с ключевого слова `module`, за которым следует объявление портов. Внутри модуля объявляются внутренние сигналы и описывается логика работы схемы, затем идет описание модуля и завершается ключевым словом `endmodule`. Внутри модуля могут создаваться экземпляры одной или нескольких копий других модулей.

В простейшем случае модуль на SystemVerilog состоит из комбинации базовых вентилей, описание которых приведено в табл. 3.1.

В качестве примера опишем простую комбинационную схему – мультиплексор 2:1. Мультиплексор имеет два информационных входа x_0 и x_1 и один адресный вход a и выход y (рис. 3.1).

Таблица 3.1

Описание базовых вентилей на языке SystemVerilog

Тип вентиля	Синтаксис	Описание
И	`and g(y, x1, x2, …)`	Выполняет логическую операцию И над двумя и более входами
ИЛИ	`or g(y, x1, x2, …)`	Выполняет логическую операцию ИЛИ над двумя и более входами
Исключающее ИЛИ	`xor g(y, x1, x2, …)`	Выполняет логическую операцию исключающее ИЛИ над двумя и более входами
И-НЕ	`nand g(y, x1, x2, …)`	Выполняет логическую операцию И-НЕ над двумя и более входами
Эквивалентность	`xnor g(y, x1, x2, …)`	Выполняет логическую операцию эквивалентность над двумя и более входами
Буфер	`buf g(y, x)`	Буферный элемент
НЕ	`not g(y, x)`	Выполняет логическую операцию НЕ
Трехстабильный буфер с управляющим входом	`bufif1 g(y, x, c)`	Буферный элемент с управляющим входом `c`. Входной сигнал `x` проходит на выход `y` только тогда, когда управляющий сигнал `c` равен 1, иначе выход устанавливается в z-состояние
Трехстабильный буфер с управляющим входом	`bufif0 g(y, x, c)`	Буферный элемент с управляющим входом `c`. Входной сигнал `x` проходит на выход `y` только тогда, когда управляющий сигнал `c` равен 0, иначе выход устанавливается в z-состояние
Трехстабильный инвертор с управляющим входом	`notif1 g(y, x, c)`	Элемент НЕ с управляющим входом `c`. Отрицание входного сигнала `x` проходит на выход `y` только тогда, когда управляющий сигнал `c` равен 1, иначе выход устанавливается в z-состояние
Трехстабильный инвертор с управляющим входом	`notif0 g(y, x, c)`	Элемент НЕ с управляющим входом `c`. Отрицание входного сигнала `x` проходит на выход `y` только тогда, когда управляющий сигнал `c` равен 0, иначе выход устанавливается в z-состояние

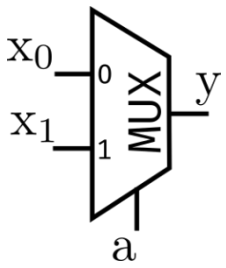

Рис. 3.1. Обозначение мультиплексора 2:1

Если адресный вход a равен нулю, то выход $y = x_0$, в противоположном случае $y = x_1$. Поведение мультиплексора описывается логической функцией

$$y = x_0\overline{a_0} + x_1 a_0.$$

Реализация мультиплексора на уровне логических вентилей показана на рис. 3.2.

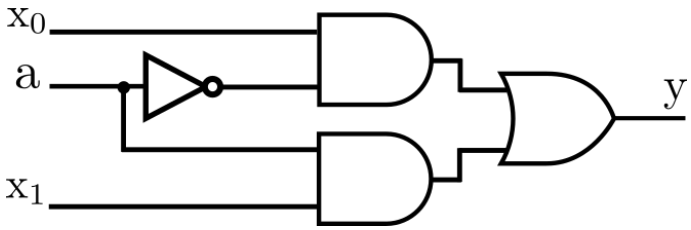

Рис. 3.2. Реализация мультиплексора в базисе логических вентилей

На языке SystemVerilog мультиплексор *на уровне вентилей* (англ. *gate-level modeling*) может быть описан следующим образом.

```
module mux
(
    input logic x0, x1, a,
    output logic y
);

    logic not_a, f1, f2;

    not g1(not_a, a);
    and g2(f1, x0, not_a);
    and g3(f2, x1, a);
    or  g4(y, f1, f2);
endmodule
```

Приведенное описание, по сути, является текстовым аналогом схемы, представленной на рис. 3.2. Для удобства сопоставления на рис. 3.3 показана схема того же мультиплексора с указанием названий внутренних связей.

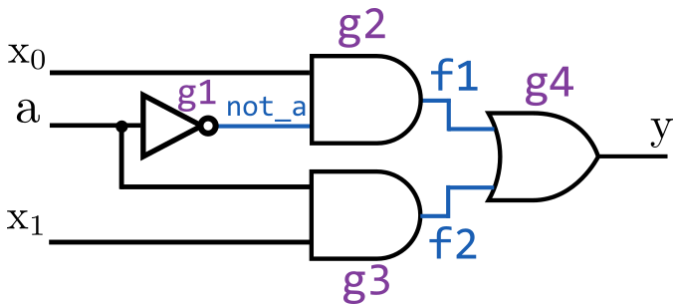

Рис. 3.3. Схема мультиплексора с обозначением внутренних связей

В дальнейшем мы будем использовать приведенное описание мультиплексора для иллюстрации некоторых концепций языка SystemVerilog.

Логические типы данных

В примере с мультиплексором входы и выходы объявлены как однобитовый тип логических данных (тип `logic`). Логический тип данных может принимать четыре различных значения, которые перечислены в табл. 3.2.

Таблица 3.2

Значения типа данных `logic`

Значение	Описание
`0`	Логический нуль
`1`	Логическая единица
`x`	Неопределенное состояние
`z`	Высокоимпедансное состояние, третье состояние

Значения `0` и `1` типа данных `logic` не требуют дополнительных пояснений, они используются для обозначения того, что внутренний сигнал схемы находится в одном из двух устойчивых состояний. Значение `X` возникает в том случае, если внутренняя линия схемы не проинициализирована. Значение `Z` возникает, когда внутренний сигнал схемы не имеет источника, т. е. им ничто не управляет. Таких ситуаций следует избегать. Состояния `X` и `Z` имеют значение только на этапе моделирования, их появление на линиях, как правило, сигнализирует разработчику о наличии ошибок в описании схемы.

Для ускорения процесса моделирования в SystemVerilog используют тип `bit`. Битовый тип принимает всего два значения: `0` и `1`, и поэтому занимает в памяти меньше места, а также упрощает анализ сигналов данного типа.

В SystemVerilog существуют и другие типы данных, из которых стоит упомянуть `int` и `tri/wire`. Тип `int` – это 32-разрядные целые числа, которые не допускают значений `X` и `Z`. Тип `int` обычно используется для описания итераций циклов при составлении тестов, но не используется для описания синтезируемых

конструкций. Линии (сигналы), имеющие тип `tri/wire`, могут управляться несколькими источниками, что используется для моделирования/описания буферов с тремя состояниями и при описании шин.

Чтобы проиллюстрировать использование типа `wire`, рассмотрим следующую схему (рис. 3.4).

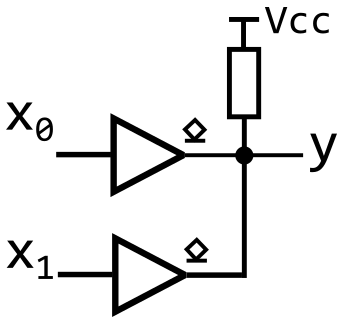

Рис. 3.4. Схема «монтажного» И

Приведенная схема реализует так называемое «монтажное» И. Здесь использованы два буфера с открытым коллектором (или открытым стоком). Если хотя бы на одном из входов будет установлен низкий уровень, то выходная линия будет замкнута на «землю». Если на двух входах установлен высокий уровень, то это приведет к тому, что на выходной линии будет присутствовать высокий уровень, т. е. логическая единица. Таким образом, представленная схема из двух буферов с объединением двух выходов реализует логическое («монтажное») И для положительной логики. Заметьте, что при использовании обычных вентилей объединение выходов схем не допускается.

Ниже приведено описание «монтажного» И на SystemVerilog.

```
module mount_and(
    input logic [1:0] x,
    output wire y
    );

buf(weak1, strong0) b1(y, x[0]);
buf(weak1, strong0) b2(y, x[1]);

endmodule
```

Обратим внимание, что при объявлении выходного порта используется тип `wire`, поскольку только данный тип позволяет описывать управление линией с двумя источниками (англ. *drivers*). При описании буферных элементов мы указываем *силу* (англ. *strength*) сигналов на входах и выходах элементов. Значение `weak1` указывает, что выход буферного элемента «подтянут к 1» (англ. *week pull-up to 1*), т. е. подключен через резистор к напряжению питания, как показано на

рис. 3.4. Такое подключение позволяет получать логическую единицу на выходе буфера при подаче на вход высокого уровня сигнала. Использование значения `strong0` при указании силы сигнала на входе говорит о том, что на входе по умолчанию подан «сильный» (стабильный) низкий логический уровень.

На рис. 3.5 показана временная диаграмма работы схемы «монтажного» И.

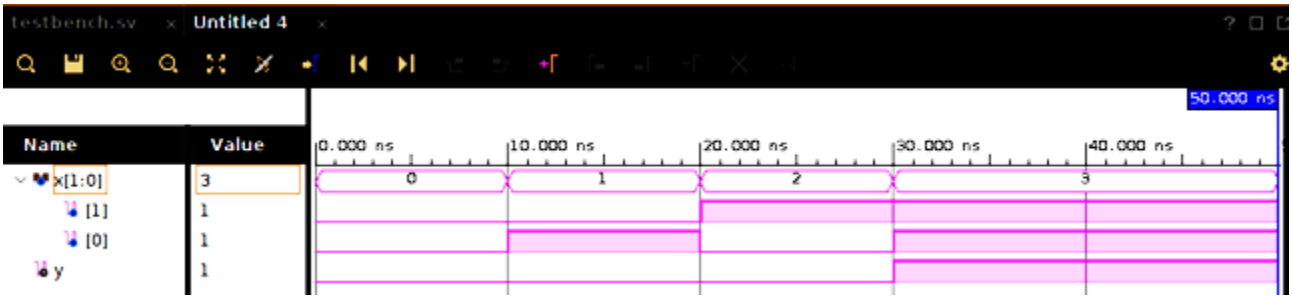

Рис. 3.5. Тестирование «монтажного» И

Рекомендуется использовать тип данных `logic` для синтезируемых конструкций, если только нет необходимости использовать другой тип данных.

Создание экземпляра модуля

В SystemVerilog цифровые схемы описываются в соответствии с принципами иерархичности и вложенности. Поэтому схемы более высокого уровня всегда включают модули более низкого уровня. Мы уже это видели, когда описывали мультиплексор (см. рис. 3.2). По сути, при описании модуля `mux` после объявления портов и внутренних сигналов мы *создали экземпляры* четырех примитивных логических элементов. Сами логические элементы мы не описывали, поскольку они являются стандартными модулями SystemVerilog. Важно понять, что создание экземпляра (англ. *instantiation*) является параллельным оператором, поэтому не важно, в каком порядке создаются модули. Создание экземпляров модулей можно сравнить с процессом рисования схемы устройства на бумаге: не важно, в каком порядке вы рисуете элементы, главное, чтобы все они были изображены, а также, чтобы все связи между ними были указаны. Конечным результатом описания является аппаратная схема со всеми связями между ее элементами.

Формально синтаксис создания экземпляра модуля на SystemVerilog следующий.

```
module_name instance_name (arg1, arg2, ...)
```

Вначале указывается название модуля (`module_name`), затем указывается название экземпляра (`instance_name`), в скобках происходит назначение сигналов на входы и выходы модуля. Название экземпляра полезно при отладке, особенно

если в проекте есть несколько экземпляров одного и того же модуля. Назначение входных/выходных портов может осуществляться различными способами:

– позиционное соответствие;

– ключевое соответствие («передача по имени»).

В примере с описанием мультиплексора `mux` назначение портов выполняется при помощи позиционного соответствия. Это стандартный стиль, который напоминает передачу аргументов в функцию в языке программирования C. При позиционном соответствии каждый сигнал передается в соответствующий порт в порядке следования портов при определении модуля. Стандартные логические элементы (`and`, `or`, `buf` и др.) всегда передают выходные данные в качестве первого порта, за которым следуют входные данные (см. табл. 3.1).

Предположим, что внутри схемы имеются сигналы X, Y, Z, W и нам необходимо создать экземпляр мультиплексора `mux` с назначением портов $x_0 = X$, $x_1 = Y$, $a_0 = Z$, $y = W$. Используя позиционное соответствие в языке SystemVerilog, мы можем сделать это следующим образом.

```
mux mux_unit (X, Y, Z, W)
```

Несмотря на простоту приведенного выше способа назначения портов, его следует избегать, отдавая предпочтение ключевому соответствию. Все дело в том, что при использовании ключевого соответствия вероятность допустить ошибку гораздо ниже. Представим себе ситуацию, что по какой-то причине разработчик внес изменение в порядок объявления портов в модуле (например, выходной порт `y` мультиплексора `mux` переместил на первое место). В этом случае необходимо проследить, чтобы во всех экземплярах модуля были внесены соответствующие правки. Если хотя бы в одном месте не выполнить переназначения, то впоследствии обнаружить такую ошибку будет достаточно сложно.

По этой причине следует использовать передачу по имени, т. е. ключевое соответствие, где каждому порту явно присваивается имя внутреннего сигнала. Пример ключевого соответствия приведен ниже.

```
mux mux_unit (.x0(X), .x1(Y), .a0(Z), .y(W))
```

Теперь мы можем изменить порядок портов так, как захотим, без необходимости обновлять экземпляры. Имена линий привязаны при помощи ключевого соответствия к соответствующим портам.

Если имя порта и имя сигнала, передаваемого в этот порт, совпадают, то круглые скобки могут быть опущены. Если в приведенном выше примере сигнал X переименовать в x_0, а Y переименовать в x_1, то назначение по имени будет выглядеть следующим образом.

```
mux mux_unit (.x0, .x1, .a0(Z), .y(W))
```

В SystemVerilog для автоматического выполнения соответствия по имени предусмотрена операция, имеющая синтаксис `.*`. Однако в этом случае трудно точно знать, какие порты определяет модуль и какие сигналы передаются.

```
mux mux_unit (.*, .a0(Z), .y(W))
```

Рекомендуется всегда использовать назначение по имени (ключевое соответствие) и избегать использования конструкции `.*`, чтобы назначение всех портов было явным.

Векторы

SystemVerilog позволяет объявлять сигналы как векторы, используя нотацию `[N:0]`. Значения в квадратных скобках определяют допустимые индексы (включая `0`) вектора. Таким образом, делая объявление `logic [1:0]`, мы создаем вектор из 2 бит, а объявляя `logic [N-1:0]`, создаем битовый вектор длиной `N`. Обращение к i-му элементу в векторе выполняется с помощью конструкции `[i]`, также можно получить доступ к фрагменту битового вектора, указав диапазон интересующих индексов `[i:j]`.

Предположим, что мы хотим описать мультиплексор 2:1, на вход которого подаются двухразрядные данные (рис. 3.6).

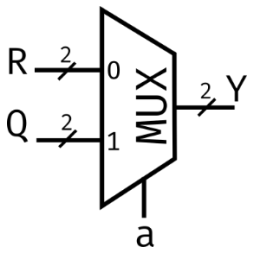

Рис. 3.6. Мультиплексор 2:1 для двухразрядных данных

На языке SystemVerilog структурное описание такого устройства, с использованием однобитных мультиплексоров 2:1, имеет следующий вид.

```
module mux_2bit(
    input logic [1:0] R, Q,
    input logic a,
    output logic [1:0] Y
);
    mux mux_1 (.x0(R[0]), .x1(Q[0]), .a(a), .y(Y[0]));
    mux mux_2 (.x0(R[1]), .x2(Q[1]), .a(a), .y(Y[1]));
endmodule
```

Схема, соответствующая приведенному описанию, показана на рис. 3.7.

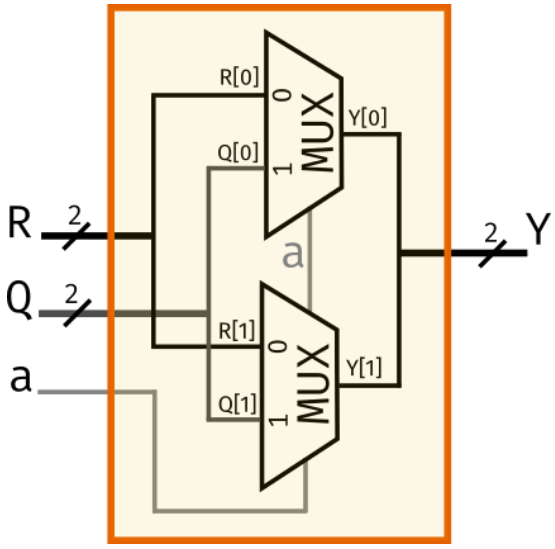

Рис. 3.7. Структура мультиплексора 2:1 для двухразрядных данных

В данном случае сигналы `R`, `Q` и `Y` объявлены как векторы с использованием нотации `[N:0]`. Обратим внимание, что, используя конструкцию `logic [1:0]`, мы получаем вектор из 2 бит. В описании на SystemVerilog экземпляр однобитного мультиплексора `mux_1` отвечает за передачу младших разрядов, а `mux_2` за передачу старших разрядов. На адресный вход подается один и тот же сигнал, чтобы обеспечить передачу на информационный выход `Y` либо входа `R`, либо входа `Q`.

3.1.3. Литералы

Литералы – это константные числовые значения, которые могут быть представлены в различных системах счисления. По умолчанию SystemVerilog поддерживает целочисленные литералы, которые представляют собой постоянные целые числа без дробной части. Для моделирования и синтеза компилятор SystemVerilog должен знать (или предполагать) ряд характеристик каждого целочисленного литерала: его размер, наличие знака, а также основания счисления. Если при задании литерала опустить данную информацию, такое значение будет интерпретироваться как 32-разрядное, знаковое и десятичное.

Основание литерала

Основание системы счисления литерала указывается апострофом, за которым следует символ, указывающий основание.

```
<разрядность>’<основание><число>
```

Чаще всего используют следующие основания:

– `b` – двоичное;
– `d` – десятичное;
– `h` – шестнадцатеричное;
– `o` – восьмеричное.

Например, значение `'b0101` будет интерпретировано как 32-разрядное беззнаковое целое число. Обратите внимание, что в данном примере указание разрядности отсутствовало и поэтому по умолчанию оно принимается равным 32. Если мы зададим литерал `4'b0101`, то он будет интерпретирован как 4-разрядное беззнаковое целое число.

Знак и разрядность литерала

Чтобы сделать литерал *знаковым*, нужно добавить символ `s` сразу после апострофа. Так, например, литерал `'sh7FF` будет интерпретирован как знаковое 32-разрядное шестнадцатиразрядное число `000007FFh`. Наличие знака влияет на арифметические и другие операции, выполняемые с литералами.

Размер, или разрядность, целочисленного литерала также может быть задан добавлением числа перед апострофом. Если литерал используется для описания синтезируемых конструкций, которые затем будут отображать на архитектуру ПЛИС, то указывать разрядность необходимо. В табл. 3.3 приведены некоторые примеры целочисленных литералов с указанием всех их характеристик.

Таблица 3.3

Примеры литералов

Значение литерала	Основание	Наличие знака	Разрядность
`8'd32`	Десятичное	Беззнаковое	8 бит
`4'sb1011`	Двоичное	Знаковое	4 бита
`16'hAE`	Шестнадцатеричное	Беззнаковое	16 бит

SystemVerilog позволяет указать для литерала большую или меньшую разрядность, чем количество бит, необходимых для представления значения. Если фактическое количество бит меньше указанной разрядности, то верхние разряды будут расширены нулями (не знаковым разрядом!). Если фактическое количество бит больше указанной разрядности, то верхние разряды будут отброшены.

Рекомендуется всегда указывать разрядность и основание литерала. Используйте только двоичное и шестнадцатеричное основания, поскольку они имеют наибольшее распространение в цифровой схемотехнике.

3.1.4. Параметры

Параметры представляют собой механизм определения констант в модуле, значение которых может определяться в момент сборки проекта.

В SystemVerilog предусмотрено два типа параметров:

– `parameter` – это константа, значение которой может быть указано при создании экземпляра модуля;

– `localparam` – внутренняя константа текущего модуля.

Первым типом часто пользуются для параметризации разрядности модуля. Самый распространенный пример – сумматор. Используя конструкцию `parameter`, можно написать модуль `N`-разрядного сумматора, в дальнейшем при создании экземпляра данного модуля можно указать разрядность входных операндов. Параметры должны быть объявлены с указанием значения по умолчанию перед определением портов, после чего на них можно ссылаться в любом месте модуля. Локальный параметр `localparam` должен быть объявлен внутри модуля и являться локальным для модуля. Приведем пример, в котором используется оба типа параметров.

```
module add_sub_module
    #(parameter N = 8)
    (
        input logic [N-1:0] A,
        input logic [N-1:0] B,
        input logic opcode,
        output logic [N-1:0] Res
    );
localparam op_add = 1'b0;
localparam op_sub = 1'b1;

always_comb begin
  if (opcode == op_add)
    Res = A + B;
  else begin
    Res = A - B;
  end
end
endmodule
```

В данном примере описан модуль, выполняющий суммирование или вычитание `N`-разрядных двоичных чисел. Выбор операции зависит от управляющего сигнала `opcode`. Если `opcode = 1`, то выполняется сложение, в противном случае модуль производит вычитание `A - B`. Код операции задан с использованием `localparam`. Из примера видно, что сложение кодируется значением `0`, а вычитание значением `1`. Назначение оператора `always_comb` будет рассмотрено позже.

Параметр `N` в приведенном примере имеет по умолчанию значение `8`. Это значение может быть изменено на этапе создания экземпляра модуля, как показано в следующем примере.

```
add_sub_module #(.N(32)) add_sub_unit(…)
```

В данном случае создается экземпляр сумматора-вычитателя на 32 разряда.

Модули могут иметь несколько параметров, поэтому в примере мы использовали для них передачу по имени, однако позиционное соответствие также поддерживается. Параметры являются мощным средством SystemVerilog, поскольку позволяют параметризировать модули, управлять разрядностью данных размеров и не только.

Некоторые литералы могут иметь размер, который определяется контекстом. В этом случае также можно присваивать значения литералам, даже если их размер параметризирован. Например, однобитовое значение `'0`, для которого не указан явно размер, будет расширено до `N` двоичных нулей, если это следует из контекста. Аналогичным образом интерпретируются `'1`, `'x`, `'z`. Например, в следующем фрагменте кода на SystemVerilog сигналу `y` присваиваются все нулевые значения.

```
logic [N-1:0] y;
assign y = '0;
```

3.1.5. Непрерывное присвоение

В данном разделе мы начнем рассматривать средства SystemVerilog, применяемые для описания цифровых устройств на уровне регистровых передач. Самым распространенным оператором данного типа является *непрерывное присвоение* (англ. *continuous assignment*) – `assign`. Оператор `assign` используется для связывания сигнала с логическим выражением. Например, вместо того чтобы создавать экземпляр вентиля И, можно воспользоваться непрерывным назначением, как показано в следующем примере.

```
assign c = a & b;
```

При любом изменении сигналов в правой части выражения сигнал в левой части будет также изменяться. С оператором `assign` можно использовать любой из логических операторов SystemVerilog, приведенных в табл. 3.4.

При помощи оператора `assign` можно описывать и более сложные комбинационные схемы. Предположим, что задана следующая схема на рис. 3.8, для которой необходимо составить код на SystemVerilog.

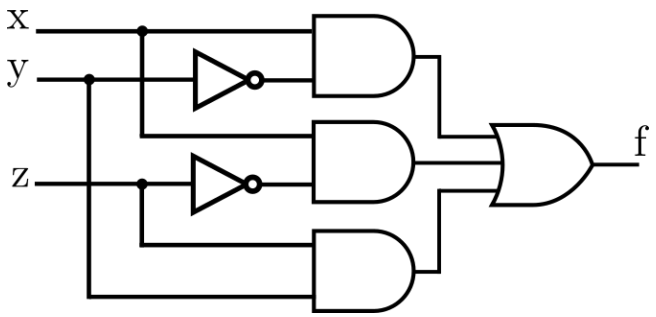

Рис. 3.8. Комбинационная схема

Таблица 3.4

Логические операторы SystemVerilog

Операция	Значение
`~a`	Операция НЕ
`a & b`	Операция И
`a | b`	Операция ИЛИ
`a ~& b`	Операция И-НЕ
`a ~| b`	Операция ИЛИ-НЕ
`a ^ b`	Исключающее ИЛИ
`a ~^ b`	Исключающее ИЛИ-НЕ

Мы можем описать комбинационную схему, записав соответствующую ей булеву функцию, используя оператор непрерывного назначения внутри модуля. Для этого необходимо указать ключевое слово `assign`, затем имя цепи или порта слева и символ назначения `=`, после чего записываем SystemVerilog-выражение с правой стороны.

```
module circuit ( output f,
                 input x, y, z);
  assign f = (x & (~y)) | (x & (~z)) | (y & z);
endmodule
```

Обратите внимание, что SystemVerilog делает те же предположения о приоритете логических операций, которые мы делаем для логических выражений. Сначала вычисляются операторы НЕ (`~`), затем операторы И, И-НЕ (`&`, `~&`), далее идут операторы исключающего ИЛИ (`^`, `~^`) и, наконец, операторы ИЛИ, ИЛИ-НЕ (`|`, `~|`). Однако мы можем включать круглые скобки в выражения SystemVerilog, чтобы уточнить или принудительно изменить порядок вычисления операторов. С учетом данного замечания описанная выше булева функция могла бы иметь следующую запись.

```
assign f = x & ~y | x & ~z | y & z;
```

Если разработчик описывает модели SystemVerilog для комбинационных схем, то, как правило, не следует пытаться переставлять логические выражения таким образом, чтобы они подразумевали какой-либо конкретный набор вентилей. Лучше выразить логические уравнения таким образом, чтобы они были наиболее понятны, а затем позволить инструментам САПР синтезировать и оптимизировать схему на основе ограничений и выбранной стратегии реализации (максимального быстродействия либо минимальной площади, например). Как правило, САПР обычно справляются с этой задачей лучше, чем это мог бы сделать разработчик вручную.

Условное присвоение

Оператор условного присвоения `:?` в SystemVerilog позволяет проверить истинность условия, заданного логическим выражением, после чего выполнить присвоение одного из двух возможных значений логической цепи.

В общем случае оператор условного присвоения можно записать следующим образом.

```
assign OUT = select ? A : B;
```

Выходному сигналу `OUT` будет присвоено значение `A`, если `select = 1`, иначе `OUT = B`, если `select = 0`.

С точки зрения схемотехники оператор условного присвоения в большинстве случаев синтезируется в виде мультиплексора. Оператор `:?` также называют тернарным оператором, поскольку он имеет три входа. Аналогичный оператор имеется в языках C и Java.

В качестве примера приведем описание двухвходового мультиплексора при помощи оператора условного присвоения.

```
module mux2(input  logic [3:0] x0, x1,
            input  logic a,
            output logic [3:0] y);
assign y = a ? x1 : x0;
endmodule
```

Если адресный вход `a = 1`, то выходной сигнал `y = x1`, в противном случае (когда `a = 0`) `y = x0`. Результат синтеза в Vivado модуля `mux2` показан на рис. 3.9.

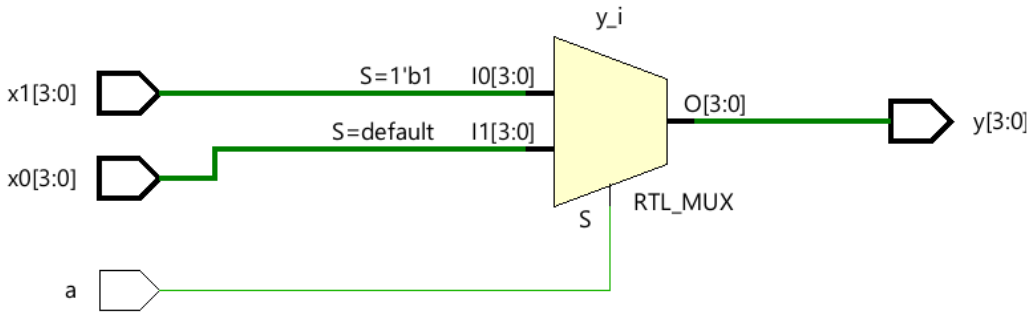

Рис. 3.9. Результат синтеза выражения с оператором условного присвоения

Можно констатировать, что Vivado распознал тернарный оператор и поставил ему в соответствие схему мультиплексора.

Поразрядные битовые операции

Битовые операции принимают на вход один или два вектора и возвращают вектор такой же длины. Например, для двух векторов `a = 01101001` и `b = 01010011` результат `a & b` будет `01000001`. Оператор И выполняется поразрядно (т. е. $c_i = a_i \cdot b_i$). Приведенные ниже два SystemVerilog-описания полностью эквивалентны (листинг 3.1).

```
assign c[0] = a[0] & b [0];
assign c[1] = a[1] & b [1];
assign c[2] = a[2] & b [2];
assign c[3] = a[3] & b [3];
```

```
assign c = a & b;
```

Листинг 3.1. VHDL-описание комбинационной схемы

SystemVerilog поддерживает поразрядные операции, приведенные в табл. 3.5.

Таблица 3.5

Поразрядные операции SystemVerilog

Операция	Значение
`~a`	Поразрядное НЕ
`a & b`	Поразрядное И
`a | b`	Поразрядное ИЛИ
`a ^ b`	Поразрядное исключающее ИЛИ

Поразрядная операция И-НЕ может быть выполнена как `~(a & b)`, а поразрядное ИЛИ-НЕ как `~(a | b)`.

Операторы сокращения

Операторы сокращения позволяют описывать многовходовые вентили, работающие с одной шиной. Оператор сокращения принимает на вход вектор логических значений и возвращает на выход один разряд. В качестве примера приведем описание восьмивходового вентиля И с входами a_7, a_6, …, a_0.

```
module and8(input logic [7:0] a,
            output logic y);
assign y = &a;
// запись &a гораздо проще, чем
// assign y = a[7] & a[6] & a[5] & a[4] & a[3] & a[2] & a[1] & a[0];
endmodule
```

SystemVerilog поддерживает операторы сокращения, приведенные в табл. 3.6.

Таблица 3.6

Операторы сокращения SystemVerilog

Операция	Значение
`& a`	Сокращение И
`| a`	Сокращение ИЛИ
`^ a`	Сокращение исключающее ИЛИ
`~& a`	Сокращение И-НЕ
`~| a`	Сокращение ИЛИ-НЕ
`~^ a`	Сокращение исключающее ИЛИ-НЕ

z-состояние

В SystemVerilog значение `z` типа `logic` используется для описания высокоимпедансного состояния. Это утверждение лучше всего проиллюстрировать, показав, как описывается буфер с тремя состояниями (рис. 3.10). Выход буфера `Y` находится в высокоимпедансном (отключенном) состоянии, когда на вход `EN` подан сигнал логического нуля. При подаче на вход разрешения `EN` логической единицы на выходе `Y` появляется значение, поданное на вход `A`. Буферные элементы используются для описания шин цифровых устройств.

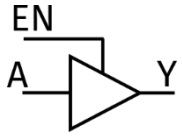

Рис. 3.10. Буфер с тремя состояниями

На языке SystemVerilog буфер с тремя состояниями описывается следующим образом.

```
module tristate_buf(input logic A,
                    input logic EN,
                    output tri  Y);
assign Y = EN ? A : 1'bz;
endmodule
```

В данном примере для выходного порта использован тип `tri`, а не `logic`. Тип `logic` не подходит, поскольку сигналы этого типа могут иметь только один драйвер. Тристабильные шины могут иметь несколько драйверов, для описания таких цепей в языке SystemVerilog предусмотрен тип `tri`. Обычно только один драйвер на шине активен в конкретный момент времени, и сигналы шины принимают задаваемое им значение. Если ни один из драйверов не активирован, то `tri` находится в высокоимпедансном состоянии (`z`). Если для входа или выхода модуля тип не указан, то по умолчанию считается, что тип – `tri`. Также нужно запомнить, что выход модуля типа `tri` может использоваться как вход модуля типа `logic`.

Арифметические операторы

Операторы, которые были рассмотрены выше, хорошо отображаются на известные в схемотехнике вентильные схемы. В данном подразделе будут рассмотрены арифметические операторы, которые относятся к синтезируемому подмножеству языка SystemVerilog. Эти операторы являются мощными средствами SystemVerilog, поскольку описать схему сумматора на вентильном уровне является нетривиальной задачей. Арифметические операторы, поддерживаемые в SystemVerilog, представлены в табл. 3.7.

Таблица 3.7

Арифметические операторы SystemVerilog

Операция	Значение
`a + b`	Сложение
`a - b`	Вычитание
`-a`	Унитарный минус
`a * b`	Умножение
`a / b`	Деление
`a % b`	Нахождение остатка от деления
`a ** b`	Возведение `a` в степень `b`

Некоторые из приведенных операций, такие как нахождение остатка от деления, умножение, а также возведение в степень, имеют достаточно сложную схемную реализацию, поэтому разработчику следует убедиться, что целевая ПЛИС или СБИС поддерживает данные операции. Обычно ПЛИС типа FPGA имеют в своем составе макроблоки, которые реализуют данные операции. Тем не менее лучше всего просмотреть отчет о задействованных ресурсах, чтобы убедиться в том, что нужные блоки действительно были использованы в процессе синтеза проекта.

Например, для следующего умножителя двухразрядных двоичных чисел.

```
module multiplier (input logic [1:0] A, B
                   output logic [3:0] C);
assign C = A * B;
endmodule
```

В результате синтеза в Vivado получена схема, приведенная на рис. 3.11. В данном случае для получения умножителя не были использованы специализированные макроблоки ПЛИС, а были задействованы смотровые таблицы (англ. *look-up table – LUT*).

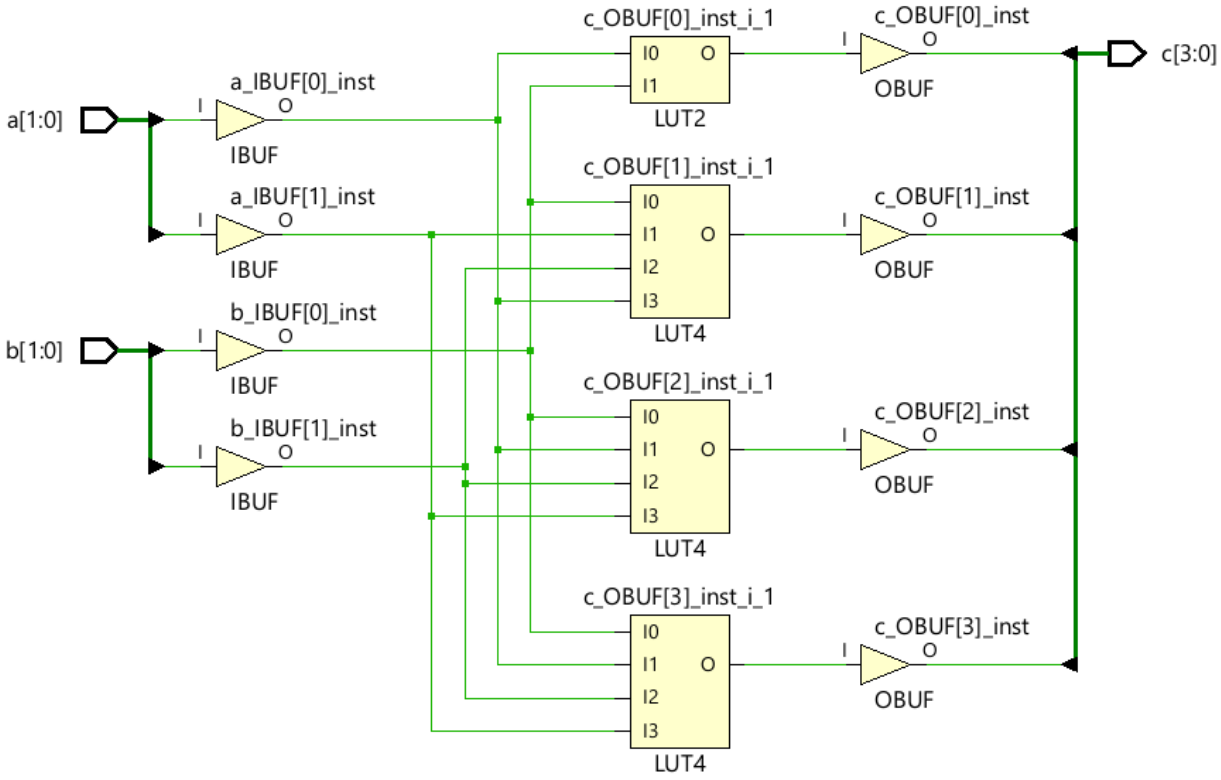

Рис. 3.11. Результат синтеза умножителя двухразрядных чисел

Операторы сдвига

Язык SystemVerilog поддерживает стандартные операции сдвига, приведенные в табл. 3.8.

Таблица 3.8

Операторы сдвига SystemVerilog

Операция	Значение
`a << b`	Логический сдвиг влево на `b` разрядов
`a >> b`	Логический сдвиг вправо на `b` разрядов
`a >>> b`	Арифметический сдвиг вправо на `b` разрядов

Арифметический сдвиг вправо сохраняет знак числа, копируя знаковый разряд в освобождающиеся разряды. Логический сдвиг вправо заполняет освободившиеся позиции нулями, аналогично действует оператор логического сдвига вправо (рис. 3.12).

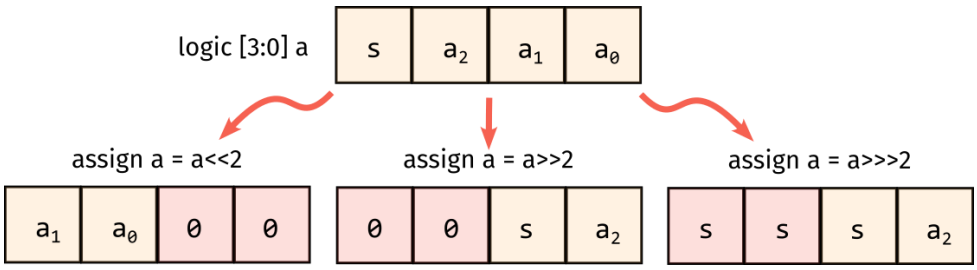

Рис. 3.12. Выполнение операций сдвига в SystemVerilog

Операторы сравнения

В SystemVerilog поддерживается много операторов сравнения (табл. 3.9).

Таблица 3.9

Операторы логического сравнения SystemVerilog

Операция	Значение
`a == b`	Проверка на равенство
`a != b`	Проверка на неравенство
`a < b`	Меньше
`a <= b`	Меньше или равно
`a > b`	Больше
`a >= b`	Больше или равно

Все приведенные операторы не требуют дополнительных пояснений. Тем не менее у операторов отношения (>, <, >=, <=) есть важная особенность – операнды в них интерпретируются как беззнаковые числа. Если не учесть этого, то можно получить неверный результат сравнения. Например, если сравнить двоичные числа

0001 > 1111, то в результате получится значение 0. Если интерпретировать числа как беззнаковые, то результат правильный, однако если в сравнение вступали знаковые числа, то результат неверный, поскольку 0001 соответствует десятичной 1, а 1111 соответствует числу -1.

При проверке на равенство/неравенство операторы == и != сравнивают только разряды, которые равны 0 или 1, и не проверяют разряды со значениями x и z. При попытке сравнить на равенство значения 1z11 и 1z11 будет возвращено значение x, т. е. неопределенность. Это является следствием того, что на аппаратном уровне значения x и z действительно нельзя сравнить. Однако при написании теста бывает полезно сравнивать все четыре значения типа logic для проверки ошибок, связанных с неинициализированными сигналами либо с неподключенными выходами. Для этих целей в SystemVerilog предусмотрены следующие два оператора (табл. 3.10).

Таблица 3.10

Операторы сравнения SystemVerilog, учитывающие значения x и z

Операция	Значение
a === b	Проверка на равенство a и b, в том числе и значений x и z
a !== b	Проверка на неравенство a и b, в том числе и значений x и z

Заметим, что операторы === и !== не относятся к синтезируемому подмножеству языка SystemVerilog, поэтому их следует применять только при написании тестов.

Также есть возможность рассматривать значения x и z как безразличные состояния при проверке на равенство. Это поведение отражено в следующих операторах (табл. 3.11), которые являются синтезируемыми конструкциями SystemVerilog.

Таблица 3.11

Операторы сравнения SystemVerilog

Операция	Значение
a ==? b	Проверка на равенство a и b, значения x и z рассматриваются как безразличное состояние
a !=? b	Проверка на неравенство a и b, значения x и z рассматриваются как безразличное состояние

Например, проверка 1z11 ==? 1x11 вернет значение логической 1, а проверка 11z1 ==? 1x11 даст значение логического 0.

Оператор конкатенации

Операторы конкатенации и репликации являются полезными для расширения векторов. Используя соответствующие операторы SystemVerilog, можно выполнить либо конкатенацию двух векторов, либо репликацию векторов (табл. 3.12).

Таблица 3.12

Операторы конкатенации и репликации

Операция	Значение
`{a,b}`	Конкатенация векторов `a` и `b`
`{n{a,b}}`	Конкатенация векторов `a` и `b` и `n`-кратное повторение (репликация) полученного вектора

Для репликации `n` должно быть задано числом (литералом) и не может быть параметром. Репликация также может быть выполнена применительно к одному вектору, в этом случае используется синтаксис `n{a}`. Данные операторы являются достаточно мощным средством SystemVerilog: с помощью репликации, например, можно выполнить знаковое расширение двоичного числа до нужного числа разрядов.

Следующий пример показывает, как можно сформировать девятиразрядный вектор из нескольких сигналов с использованием приведенных выше операторов. Допустим, что в результате должен получиться вектор со следующей структурой $b_2b_1b_0b_0b_0a_1a_001$.

```
assign y = {b[2:1], {3{b[0]}}, a[1:0], 2'b01};
```

Моделирование задержек

С каждым оператором непрерывного присвоения может быть ассоциирована задержка, указанная в единицах времени. Единица времени, используемая по умолчанию, задается в начале описания модуля при помощи директивы компилятора `timescale`. Рассмотрим следующий модуль, описывающий вентиль И.

```
'timescale 1ns/1ps

module and_gate(
    input logic A, B,
    output logic Y
);

    assign #5 y = A & B;
endmodule
```

В данном примере в качестве базовой единицы времени устанавливается 1 нс, а 1 пс – это шаг модельного времени. В операторе `assign` после знака `#` указана задержка на срабатывание. Таким образом, выход `y` будет изменяться спустя 5 нс после изменения `A` либо `B`.

3.1.6. Оператор `generate`

В некоторых случаях бывает необходимо создать заданное число экземпляров данного модуля, обычно это число задается в виде параметра. Оператор `generate` языка SystemVerilog как раз используется для создания таких конфигурируемых структур. Синтаксис данного оператора показан ниже, он может быть использован как при создании экземпляров модулей, так и совместно с оператором непрерывного присвоения.

```
generate
genvar [ индексные переменные ];
for (...; ...; ...) begin [: дополнительные метки ]
... параллельные операторы ...;
end
endgenerate
```

Рассмотрим пример использования `generate` для создания `N`-разрядного компаратора. Для этой цели можно было бы использовать оператор сокращения исключающее ИЛИ-НЕ `~^`, однако в данном примере эта же цель будет достигнута с использованием оператора `generate` и созданием заданного числа экземпляров вентиля исключающее ИЛИ-НЕ.

```
module eq_n
    #( parameter N = 4)
    (
    input logic [N -1:0] a, b,
    output logic eq
    );

    logic [N -1:0] tmp;

    generate
        genvar i;
        for (i = 0; i < N; i = i + 1)
            xnor gen_u (tmp[i], a[i], b[i]);
    endgenerate

    assign eq = & tmp ;
endmodule
```

Полученное описание соответствует схеме, приведенной на рис. 3.13.

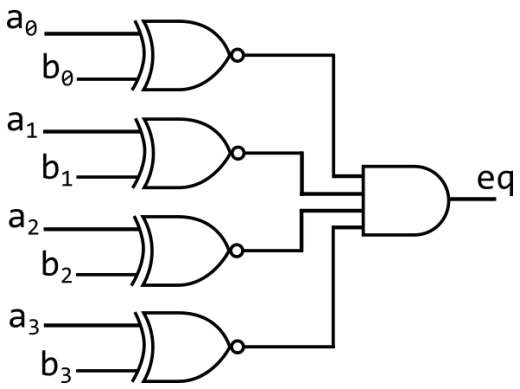

Рис. 3.13. Регулярная схема компаратора для `N = 4`

Таким образом, оператор `generate` позволяет компактно описывать регулярные схемы. По сути, он является заменой операции копирования/вставки в коде большого числа повторяющихся модулей. С его помощью можно создавать обобщенные схемы, которые зависят от разрядности (параметра `N`) входных/выходных данных.

3.1.7. Процедурный блок `always`

Оператор `always` языка SystemVerilog является мощным инструментом моделирования схем. Он соответствует оператору `process` в VHDL. Оператор `always` является основным для описания последовательностных схем, однако его также можно использовать для описания комбинационной логики. Чтобы дать компилятору понять намерение разработчика описать комбинационную схему, используется специальная версия оператора – `always_comb`. Для описания последовательностных схем подходит общая форма оператора `always`, а также две специализированные версии: `always_ff` и `always_latch`.

Оператор `always_comb` имеет следующий синтаксис.

```
always_comb begin
    [ объявление дополнительных локальных переменных ]
    ... процедурные операторы
end
```

Данный оператор является параллельным, он запускается всегда, когда изменяется состояние какого-либо входного сигнала. О нем можно мыслить, как о многострочной версии оператора непрерывного назначения. Ключевые слова `begin` и `end` являются необязательными, если внутри оператора `always_comb` есть только один процедурный оператор. Заметим, что один процедурный оператор может быть и многострочным, таким, как, например, оператор `case`. Тем не менее стоит быть очень аккуратным при опускании ключевых слов `begin` и `end`, поскольку всегда есть риск забыть их добавить при решении добавить в оператор `always_comb` дополнительные процедурные операторы.

Язык SystemVerilog поддерживает большое количество процедурных операторов (англ. *procedural statement*), которые могут быть использованы внутри оператора `always`. Многие из них не имеют четкой физической интерпретации и не могут быть синтезированы. В дальнейшем будут рассмотрены следующие четыре типа синтезируемых процедурных операторов:

– оператор блокирующего присвоения;

– оператор неблокирующего присвоения;

– условный оператор `if`;

– оператор выбора `case`.

Использование процедурных операторов допустимо только внутри процедурных блоков – `initial` и `always`. Они называются процедурными, поскольку их выполнение внутри блоков происходит последовательно. Получаемый SystemVerilog-код синтезируется в аппаратуру, которая должна моделировать последовательную семантику процедурных операторов, несмотря на то, что аппаратные блоки сами по себе не ведут себя как последовательные программы.

Ниже приведен пример простого сумматора-вычитателя, описанного с использованием оператора `always_comb`.

```
module add_sub
    (input logic [3:0] A, B,
     input logic add_en,
     output logic [3:0] Y;
     );
     always_comb
          if (add_en)
               Y = A + B;
          else
               Y = A - B;
endmodule
```

Оператор `always_comb` выполняется каждый раз, когда изменяется любой из сигналов в правой части операторов `<=` или `=` внутри процедурного блока. В данном примере выход `Y` перерассчитывается каждый раз при любом изменении входных сигналов `A`, `B` или `add_en`. Аналогичную схему можно также описать, используя обычный оператор `always`, как показано в примере ниже.

```
module add_sub
    (input logic [3:0] A, B,
     input logic add_en,
     output logic [3:0] Y);

     always @(A, B, add_en)
          if (add_en)
               Y = A + B;
          else
               Y = A - B;
endmodule
```

После знака @ в скобках указан список чувствительности оператора always. Таким образом, использование оператора always_comb гораздо надежнее, поскольку позволяет избежать ошибок в случае переименования или добавления сигналов в процедурный блок always. И в первом, и во втором приведенных примерах синтезатор Vivado сгенерирует схему, показанную на рис. 3.14.

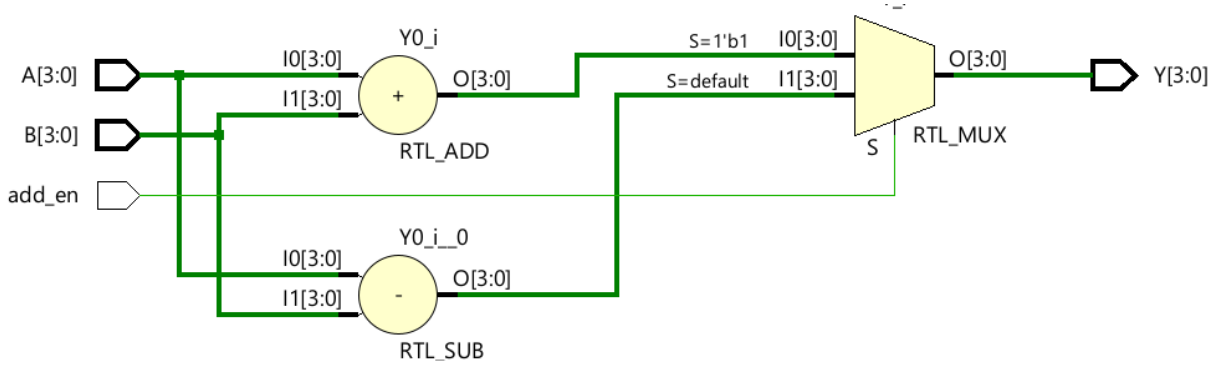

Рис. 3.14. Результат синтеза сумматора-вычитателя, описанного с использованием процедурного блока always_comb

Можно заметить, что в данном случае синтезатором на аппаратном уровне были реализованы как сумматор, так и вычитатель, а при помощи мультиплексора, управляемого сигналом add_en, осуществляется выбор того, что будет подано на выход Y – результат сложения или вычитания.

Несмотря на то что оператор always_comb предназначен для описания комбинационной логики, при его использовании могут возникнуть неопределенные ситуации. Это связано с тем, что схемы комбинационной логики по определению не имеют внутренней памяти (т. е. не сохраняют внутреннее состояние), а их выход зависит в любой момент времени только от входа. Рассмотрим следующий пример модуля, в котором удалена ветка else.

```
always_comb
    if (add_en)
        Y = A + B;
```

Что в этом случае должно быть выходом схемы при низком уровне на входе add_en? Очевидно, что Y должно хранить то состояние, которое было на выходе в момент предыдущего высокого уровня add_en. Однако такое поведение уже не соответствует комбинационной логике, и для его реализации в схему требуется добавить регистр-защелку (англ. *latch*). Но поскольку для описания использовался оператор always_comb, то синтезатор выдаст предупреждение о непредусмотренном поведении. Таким образом, для моделирования комбинационной логики необходимо, чтобы SystemVerilog-описание определяло выход для каждой комбинации входных сигналов. В приведенном примере не описано поведение Y при низком уровне на входе add_en, что в результате приводит к некомбинационной схеме.

Блокирующее назначение

Блокирующее назначение (англ. *blocking assignment*) используется только внутри процедурных блоков (например, `always` или `initial`) и имеет следующий синтаксис.

```
[ имя_переменной ] = [ выражение ];
```

При блокирующем назначении выражение вычисляется и мгновенно назначается переменной, указанной в левой части. Такое поведение полностью соответствует работе с переменными в языке C.

Блокирующее назначение используется для описания комбинационных схем, в то время как неблокирующее назначение `<=` используют для описания последовательностных схем. Неблокирующее назначение будет рассмотрено ниже.

Рассмотрим пример описания комбинационной схемы для поиска максимального значения среди трех чисел, в котором используется оператор блокирующего присвоения (листинг 3.2).

```
module find_max
   (input logic [3:0] X0, X1, X2,
    output logic [3:0] Y);

    always_comb begin
        logic [3:0] tmp; // локальная переменная
        tmp = (X1>X0)? X1 : X0;
        Y = (tmp>X2)? tmp : X2;
    end
endmodule
```

Листинг 3.2. SystemVerilog-описание модуля поиска максимума

Внутри блока `always_comb` объявлена локальная переменная `tmp`, которая затем используется в операторе условного присвоения. За счет использования блокирующего присвоения переменная `tmp` получает свое значение мгновенно и поэтому может быть использована уже в следующем процедурном операторе. Схема, синтезированная по данному описанию, действительно является комбинационной (рис. 3.15).

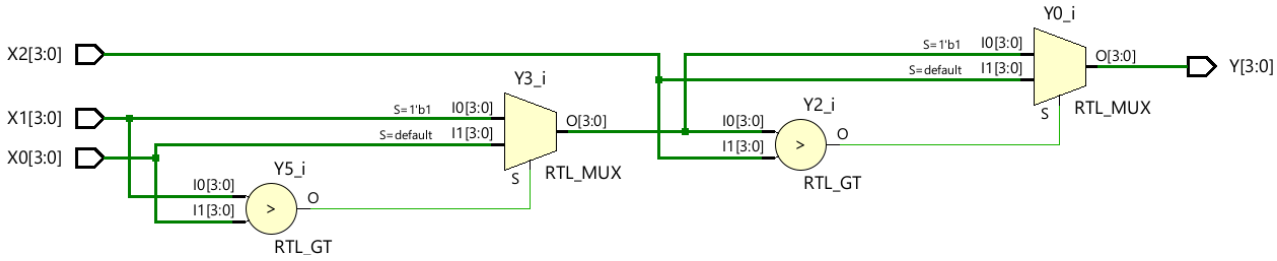

Рис. 3.15. Схема поиска максимального значения

Условный оператор `if`

Условный оператор `if` в SystemVerilog ведет себя так же, как и в большинстве языков программирования, и имеет следующий синтаксис.

```
if ([логическое_выражение]) begin
... процедурные операторы ...
end [else begin
... процедурные операторы ...
end]
```

Верхняя ветка оператора `if` вычисляется, если `[логическое_выражение]` принимает значение логической `1`, в противном случае вычисляется ветка `else`. Следует отметить, что ветка `else` также может быть пропущена. Ранее уже приводились примеры использования данного оператора в процедурном блоке `always`. Проанализировав их, можно сделать вывод, что чаще всего оператор `if` синтезируется компилятором в виде мультиплексора.

Оператор выбора `case`

Оператор множественного выбора `case` в SystemVerilog имеет следующий вид.

```
case [выражение]
    [значение_1]: begin
    ... процедурные операторы ...
    end
    [значение_2]: begin
    ... процедурные операторы ...
    end
...
    [значение_n]: begin
    ... процедурные операторы ...
    end
    [default : begin
    ... процедурные операторы ...
    end]
endcase
```

Оператор case вычисляет [выражение] и сравнивает его с каждым перечисленным значением. Если совпадение найдено, то происходит выполнение соответствующих процедурных операторов. Секция default запускается в том случае, если совпадения с предыдущими значениями не были обнаружены. Это особенно важно учитывать при написании конструкций, которые в дальнейшем должны быть реализованы в виде комбинационной логики. Использование секции default позволяет синтезатору просчитать выходы для всех возможных комбинаций входных сигналов. В случае когда выражение удовлетворяет нескольким значениям, будет вычислена только одна секция процедурных операторов, которая встретится первой. Это поведение соответствует работе приоритетного шифратора.

В качестве примера использования оператора case ниже приведено описание мультиплексора 4:1.

```
module mux4_1
   #(parameter N = 2)
   (input logic [N -1:0] X0, X1, X2, X3,
    input logic [1:0] Addr,
    output logic [N -1:0] Y);

    always_comb begin
    case (Addr)
        2'b00: Y = X0;
        2'b01: Y = X1;
        2'b10: Y = X2;
        2'b11: Y = X3;
        default: Y = X0;
    endcase
    end
endmodule
```

Неблокирующее назначение

Оператор неблокирующего присвоения <= используется для описания последовательностных синхронных схем совместно с оператором always, который в общем виде имеет следующий синтаксис.

```
always @ (... список чувствительности ...) begin
[ объявление локальных переменных ];
... процедурные операторы ...
end
```

Список чувствительности – это сигналы, изменение которых запускает выполнение процедурного блока `always`. В списке чувствительности перед сигналом может быть указано ключевое слово `posedge` или `negedge`. Указание любого из данных слов приведет к тому, что блок `always` будет запускаться либо по переднему фронту (`posedge`), либо по заднему фронту, т. е. спаду (`negedge`). Понятия переднего и заднего фронта проиллюстрированы на рис. 3.16.

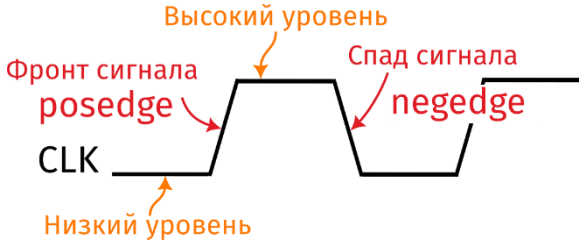

Рис. 3.16. Фронт и спад сигнала

Ключевые слова `posedge` и `negedge` используются для описания синхронных триггеров, которые переключаются в момент изменения синхросигнала.

В списке чувствительности блока `always` также может быть указан знак *. Это означает, что данный блок будет реагировать на изменение любого входного сигнала, к которому производится доступ внутри блока. Такое поведение соответствует комбинационным схемам, которые немедленно реагируют на любое изменение входных сигналов. Вместо конструкции `always@(*)` можно использовать ее сокращенную версию `always@*`. В языке SystemVerilog именно вместо данной конструкции был введен отдельный оператор `always_comb`, которого нет в языке Verilog. Однако между данными операторами есть важное отличие в том, что `always_comb` дает синтезатору дополнительную информацию о намерении разработчика получить комбинационную схему, и если эта цель не будет достигнута, то синтезатор выдаст соответствующее предупреждение.

Неблокирующее назначение имеет следующий синтаксис.

```
[ имя_переменной ] <= [ выражение ];
```

При неблокирующем присвоении выражение вычисляется, но присвоение происходит только в конце блока `always`, следовательно, данное присвоение не блокирует выполнение последующих операторов.

Самым распространенным способом хранения данных является использование D-триггеров (англ. *D flip-flop*). В простейшем случае такой триггер имеет информационный вход `D`, вход синхронизации `clk`, а также выход `Q` (рис. 3.17). Информация с входа `D` запоминается при наступлении переднего фронта сигнала `clk` и подается на вход `Q`.

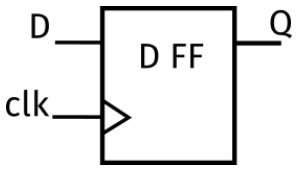

Рис. 3.17. Обозначение D-триггера

Для описания поведения D-триггера необходимо указать, что присвоение значения `D` выходу `Q` будет происходить только по переднему фронту `clk`. Для этой цели необходимо в списке чувствительности указать ключевое слово `posedge` перед сигналом `clk`, а затем внутри блока выполнить назначение `D <= Q`. В SystemVerilog есть специальное ключевое слово `always_ff`, которое рекомендуется использовать для описания синхронных триггерных структур наподобие D-триггеров. Использование `always_ff` передает синтезатору намерение разработчика получить триггерную структуру, и если эта цель не достигнута (например, вместо триггера синтезирована защелка), то синтезатор обязательно выдаст соответствующее предупреждение.

В качестве примера использования блока `always_ff` ниже приведено описание простого D-триггера на SystemVerilog.

```
module D_FF
   (input logic clk ,
    input logic D,
    output logic Q);

    always_ff @( posedge clk )
         Q <= D;
endmodule
```

Чтобы получить D-триггер с сигналом сброса, следует дополнить логику работы блока `always_ff`, как показано в примере ниже.

```
module DFF_with_reset(
    input logic D, rst, clk,
    output logic Q);

    always_ff @(posedge clk) begin
        if (rst)
            Q <= 1'b0;
        else
            Q <= D;
    end
endmodule
```

Рассмотрим, в чем заключается разница между блокирующим и неблокирующим назначениями. Для этого обратимся к примеру кода на SystemVerilog (листинг 3.3), который по замыслу должен обеспечивать задержку входного сигнала на два такта синхроимпульса.

```
module delay2
   (input logic D,
    input logic clk,
    output logic Q2);

    logic Q_tmp;

    always_ff @(posedge clk) begin
        Q_tmp = D;  // блокирующее назначение
        Q2 = Q_tmp; // блокирующее назначение
    end
endmodule
```

Листинг 3.3. SystemVerilog-описание модуля задержки

Однако поскольку внутри блока `always_ff` используется блокирующее назначение, которое мгновенно выполняет присвоение, то такое описание будет воспринято синтезатором как одиночный D-триггер и эффекта от использования промежуточной переменной `Q_tmp` не будет. Схема, синтезированная Vivado по данному описанию, показана на рис. 3.18.

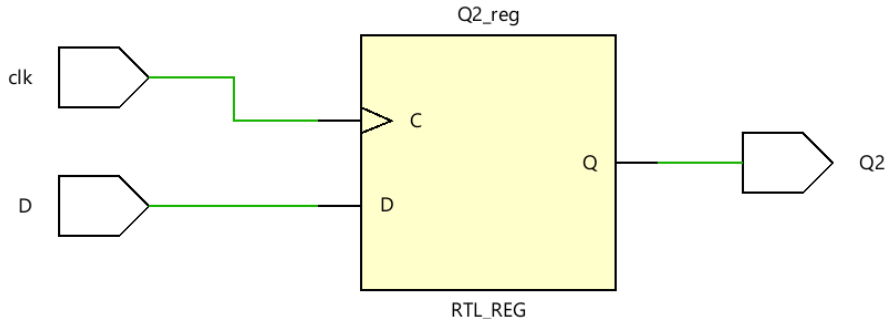

Рис. 3.18. Результат синтеза описания модуля задержки, приведенного в листинге 3.3

Ситуация изменится, если внутри блока `always_ff` воспользоваться неблокирующим назначением.

```
always_ff @(posedge clk) begin
    Q_tmp <= D;  // неблокирующее назначение
    Q2 <= Q_tmp; // неблокирующее назначение
end
```

Как было сказано ранее, в этом случае вычисления выражения в левой части происходят сразу, однако присвоение новых значений всем переменным выполняется одновременно в конце блока `always`. Такое поведение соответствует работе двух последовательно включенных D-триггеров. На рис. 3.19 показана схема, синтезированная Vivado для приведенного описания.

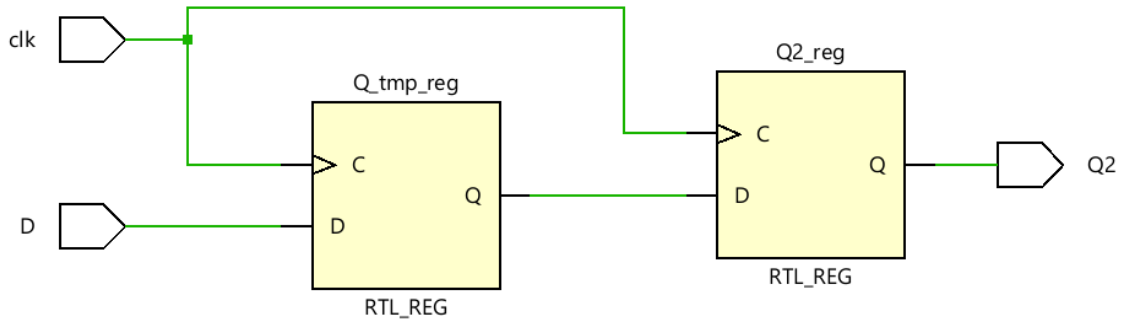

Рис. 3.19. Результат синтеза при использовании в SystemVerilog-описании модуля задержки неблокирующих назначений

3.1.8. Написание теста

Для запуска процесса моделирования разработанных SystemVerilog-описаний, как и в случае с VHDL, требуется разработка модуля с тестирующей программой (тестовое окружение). Тестовое окружение для проектов на SystemVerilog строится так же, как и для VHDL-проектов (см. рис. 1.11).

В начале модуля с тестом обязательно указывается масштаб времени для работы с симулятором. Синтаксис этой операции следующий.

```
'timescale <единица времени>/<временной шаг>
```

При описании моделирования задержек вентилей в SystemVerilog эта директива уже описывалась. При указании единиц времени можно использовать значения, приведенные в табл. 3.13.

Стандартным значением масштаба времени считается 1 нс / 10 пс.

Часто для создания теста используют процедурный блок `initial`. Это однократно протекающий процесс, запускаемый в начальный момент модельного времени. Блок `initial` может также использоваться для инициализации начальных значений элементов памяти цифровых устройств. Однако наиболее часто `initial` используют для формирования подаваемых на вход схемы тестовых сигналов при разработке тестового окружения.

Таблица 3.13

Единицы времени в SystemVerilog

Обозначение	Сокращение (на русском)	Единица измерения
s	с	Секунда
ms	мс (10^{-3} с)	Миллисекунда
us	мкс (10^{-6} с)	Микросекунда
ns	нс (10^{-9} с)	Наносекунда
ps	пс (10^{-12} с)	Пикосекунда
fs	фс (10^{-15} с)	Фемтосекунда

В качестве примера рассмотрим тест устройства поиска максимума среди трех чисел (см. листинг 3.2).

```
timescale 1ns / 10ps

module testbench;
    logic [3:0] x0, x1, x2, y; // внутренние сигналы

    find_max DUT(.X0(x0), .X1(x1), .X2(x2), .Y(y)); // создание экземпляра

    initial begin
        x0 = 4'b0001;
        x1 = 4'b1111;
        x2 = 4'b1001;
        #10; // задержка 10 нс

        x0 = 4'b0011;
        x1 = 4'b0111;
        x2 = 4'b1100;
        #10; // задержка 10 нс

        x0 = 4'b0111;
        x1 = 4'b0110;
        x2 = 4'b0100;
        #10; // задержка 10 нс
    end;
endmodule
```

В модуле, описывающем тестовое окружение, нет входных и выходных сигналов, все сигналы являются внутренними. Вначале создается экземпляр тестируемого устройства и на соответствующие порты производится назначение внутренних сигналов. В данном примере использовано ключевое соответствие портов. Далее следует блок `initial`, внутри которого формируются тестовые сигналы. С помощью оператора `#10` указывается, что между тестовыми комбинациями входных сигналов должна быть задержка в 10 нс. В результате запуска теста в симуляторе

Vivado формируется временная диаграмма, показанная на рис. 3.20. Тест показывает, что устройство функционирует правильно, каждый раз на выходе появляется максимальное из трех подаваемых значений.

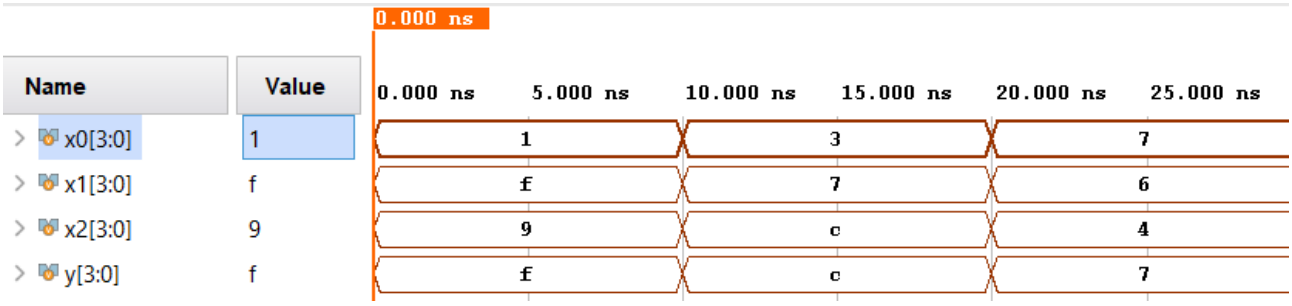

Рис. 3.20. Временная диаграмма теста устройства поиска максимального значения

Разработанный тест можно сократить, если использовать в нем цикл `for`, как показано в следующем примере.

```
timescale 1ns / 10ps

module testbench;
logic [3:0] x0, x1, x2, y;

find_max DUT(.X0(x0), .X1(x1), .X2(x2), .Y(y));

initial begin
    $printtimescale(testbench);
    for (int i=0; i<3; i++) begin
        $display("Iteration %2d", i[1:0]);
        x0 = $random();
        x1 = $random();
        x2 = $random();
        #10;
    end
    $display("Test passed!");
    $stop;
end
endmodule
```

Листинг 3.4. Тестовое окружение проекта на SystemVerilog с использованием цикла `for`

При помощи команды `$printtimescale(testbench)` в консоль отображаются единицы времени, используемые в тестовом окружении. Команда `$display("Iteration %2d", i[1:0])` используется для отображения текущего номера итерации цикла. В следующих трех строках внутренним сигналам `x0`, `x1` и `x2` назначаются случайные значения при помощи команды `$random()`, после чего выполняется задержка на 10 нс. По окончании цикла из трех итераций в консоли будет

выведена строка «Test passed!». Системная функция $stop приводит к остановке процесса моделирования. Временная диаграмма теста, приведенного в листинге 3.4, будет иметь вид, аналогичный показанному на рис. 3.20, за исключением того, что на входы x0, x1 и x2 будут подаваться случайные числа. Вместо системной функции $random(), которая возвращает 32-битное случайное число, также можно было использовать $urandom_range(a,b), которая возвращает случайное число в промежутке между a и b.

Недостатком разработанного теста является то, что для проверки корректности работы устройства необходимо просмотреть всю временную диаграмму и убедиться, что на всех наборах выходное значение выдается корректно. Для небольшого проекта это приемлемо, но для сложных систем это становится проблематичным. Чтобы автоматизировать этот процесс, можно в тест, приведенный в листинге 3.4, добавить блок проверки, показанный ниже.

```
always @(y) begin
    logic [3:0] y_groud_true;
    y_groud_true = (x0>x1) ? x0: x1;
    y_groud_true = (x2>y_groud_true)? x2 : y_groud_true;
    $display("Time = %0d, x0=%0d, x1=%0d, x2=%0d, y=%0d,", $time, x0, x1, x2, y);
    if (y!=y_groud_true)
        $info("Y value wrong!");
    else
        $info("Y value correct!");
end
```

Блок проверки реагирует на изменение выходного сигнала устройства y. Внутри блока вычисляется ожидаемое значение y, которое затем сравнивается с фактическим. В зависимости от результата сравнения в консоль выводится информационное сообщение при помощи системной функции $info о том, корректно значение y или нет.

После запуска всего теста в симуляторе Vivado в окне Tcl Console будут выведены следующие сообщения.

```
Timescale of (testbench) is 1ns/1ps.
Iteration  0
Time = 0, x0=4, x1=1, x2=9, y=9,
Info: Y value correct!
Iteration  1
Time = 10, x0=3, x1=13, x2=13, y=13,
Info: Y value correct!
Iteration  2
Time = 20, x0=5, x1=2, x2=1, y=5,
Info: Y value correct!
Test passed!
```

Прочитав выведенные сообщения, легко убедиться, что во всех трех случаях моделируемое устройство сработало правильно. Таким образом, добавление в тестовое окружение блока самопроверки значительным образом упрощает процесс отладки проекта и поиска ошибок.

3.2. Порядок выполнения работы

1. Разработать схему цифрового устройства в соответствии с вариантом (табл. 3.14).

Таблица 3.14

Варианты заданий

Номер варианта	Описание устройства
1	Мультиплексор 8:1, использующий в качестве базового элемента мультиплексор 2:1, построенный в базисе элементов И-НЕ
2	Двоично-десятичный дешифратор (рис. 3.21) в базисе элементов И-НЕ, ИЛИ-НЕ
3	Десятичный шифратор (рис. 3.22) в базисе элементов ИЛИ-НЕ, НЕ
4	Демультиплексор 1:16 в базисе четырехвходовых элементов И-НЕ
5	Дешифратор семисегментного индикатора в базисе двухвходовых элементов И-НЕ, ИЛИ-НЕ (рис. 3.23)
6	Приоритетный шифратор 16:4 в базисе элементов исключающее ИЛИ
7	Мультиплексор 16:1, использующий в качестве базового элемента мультиплексор 4:1, построенный в базисе элементов И-НЕ
8	Четырехразрядный мультиплексор 2:1 в базисе трехвходовых элементов И-НЕ, ИЛИ-НЕ

2. Выполнить структурное иерархическое описание разработанной схемы на языке SystemVerilog.

3. Разработать поведенческое описание заданного цифрового устройства на уровне регистровых передач.

4. Разработать тестовое окружение для разрабатываемого цифрового устройства, привести временную диаграмму работы устройства.

5. Подготовить отчет по лабораторной работе. В отчете должна быть представлена разработанная в п. 1 схема устройства в базисе заданных элементов, должен содержаться SystemVerilog-код схемы и тестирующая программа, должны быть представлены временные диаграммы, подтверждающие корректность работы структурного и поведенческого описания схемы.

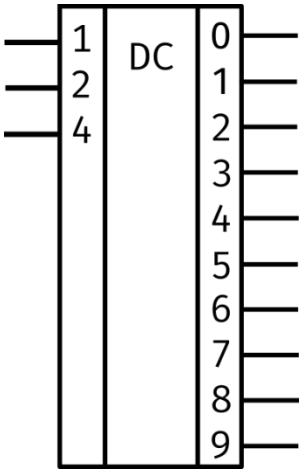

Рис. 3.21. Вариант 2 (двоично-десятичный дешифратор)

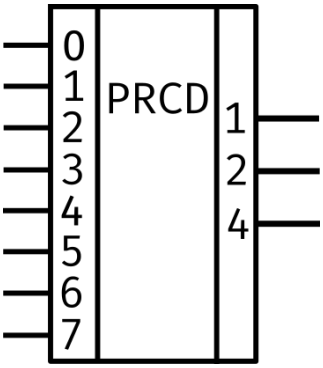

Рис. 3.22. Вариант 3 (приоритетный шифратор)

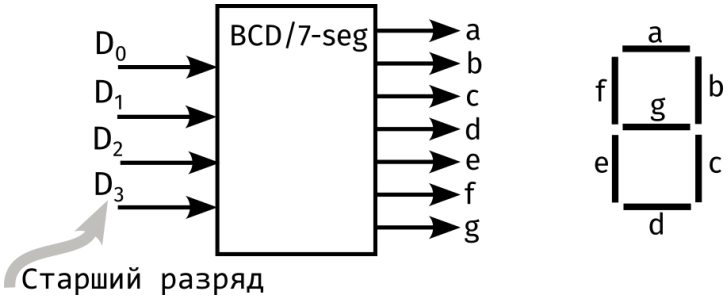

Рис. 3.23. Вариант 5 (дешифратор семисегментного индикатора)

3.3. Дополнительные вопросы и задания

1. На SystemVerilog составить описание схемы, реализующей «монтажное» ИЛИ. Написать тест, продемонстрировать корректность работы схемы.

2. Показать, как можно на SystemVerilog описать четырехвходовой мультиплексор, используя только операторы условного присвоения.

3. В операторе `assign #5 y = A & B` указывается задержка на срабатывание (5 нс). Выполнить эксперимент и проверить, является ли данная задержка транспортной или инерционной.

4. В чем разница между блокирующим и неблокирующим присвоением?

ПРИЛОЖЕНИЕ

Варианты для лабораторной работы № 2

Таблица П.1

Логические элементы

Имя элемента	Функция элемента	Задержка, нс
GND	$y = 0$	1
VCC	$y = 1$	1
N	$y = \overline{A}$	1
A2	$y = AB$	2
A3	$y = ABC$	3
A4	$y = ABCD$	4
A6	$y = ABCDEF$	6
A8	$y = ABCDEFGH$	8
NEX2	$y = AB \vee \overline{AB}$	5
NMX2	$y = \overline{(A \vee \overline{V})(B \vee V)}$	6
NO2	$y = \overline{A \vee B}$	3
NO3	$y = \overline{A \vee B \vee C}$	4
NO3A2	$y = \overline{A \vee B \vee DC}$	5
NO4	$y = \overline{A \vee B \vee C \vee D}$	5
NOA2	$y = \overline{A \vee BC}$	3
NOA22	$y = \overline{AB \vee CD}$	4
NOA3	$y = \overline{A \vee BCD}$	5
NOAO2	$y = \overline{A \vee B(C \vee D)}$	4
EX2	$y = A\overline{B} \vee \overline{A}B$	5
MX2	$y = \overline{V}B \vee VA$	3
NA2	$y = \overline{AB}$	2
NA3	$y = \overline{ABC}$	3
NA3O2	$y = \overline{AB(C \vee D)}$	4
NA4	$y = \overline{ABCD}$	5
NAO2	$y = \overline{A(B \vee C)}$	3
NAO22	$y = \overline{(A \vee B)(C \vee D)}$	3
NAO3	$y = \overline{A(B \vee C \vee D)}$	5
NAOA2	$y = \overline{A(B \vee (CD))}$	4
O2	$y = A \vee B$	2
O3	$y = A \vee B \vee C$	3
O4	$y = A \vee B \vee C \vee D$	4
O6	$y = A \vee B \vee C \vee D \vee E \vee F$	6
O8	$y = A \vee B \vee C \vee D \vee E \vee F \vee G \vee H$	8
NMX4	$y = \overline{AV_1V_2} \vee \overline{BV_1}V_2 \vee \overline{C}V_1\overline{V_2} \vee \overline{D}V_1V_2$	8

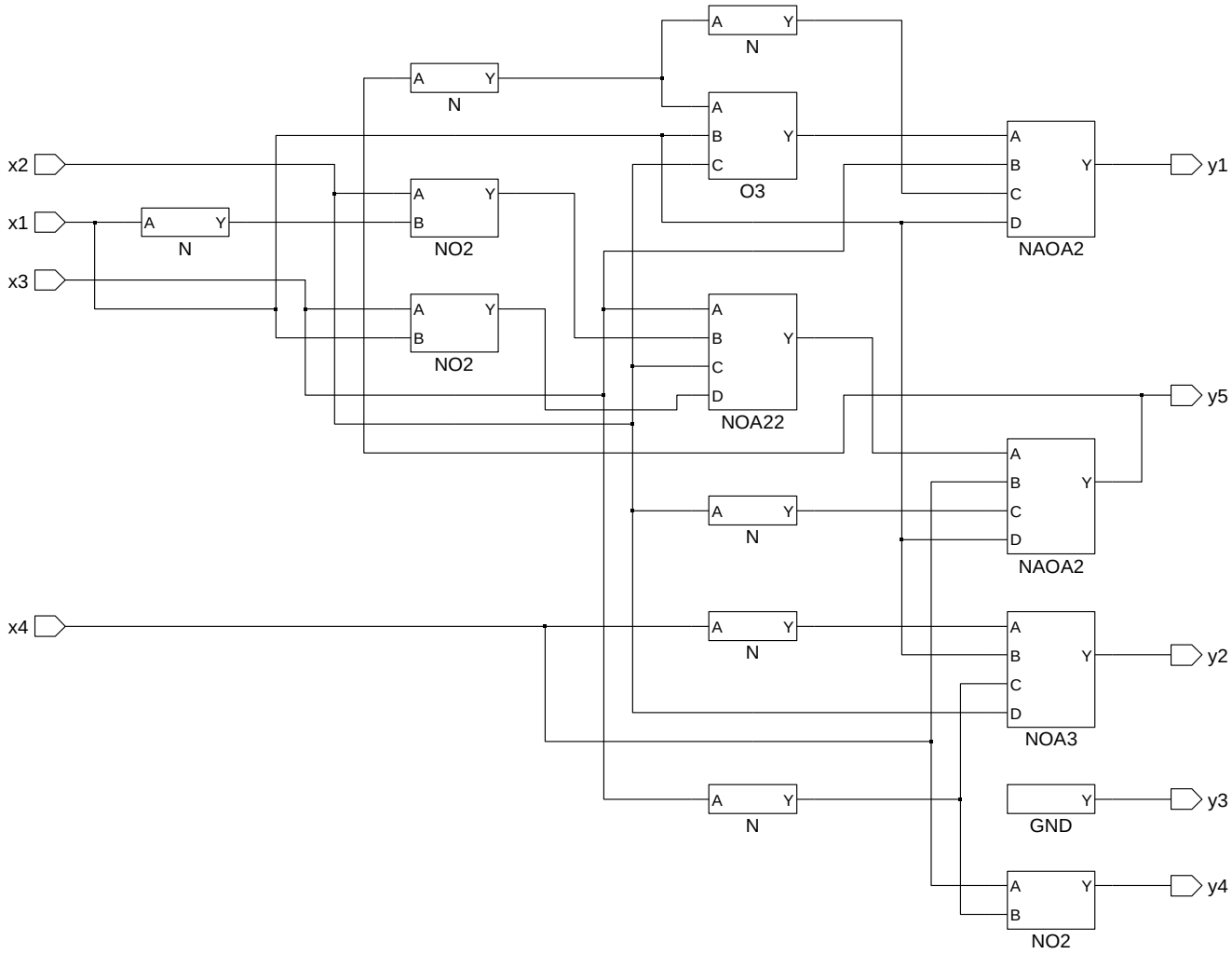

Рис. П.1. Вариант 1

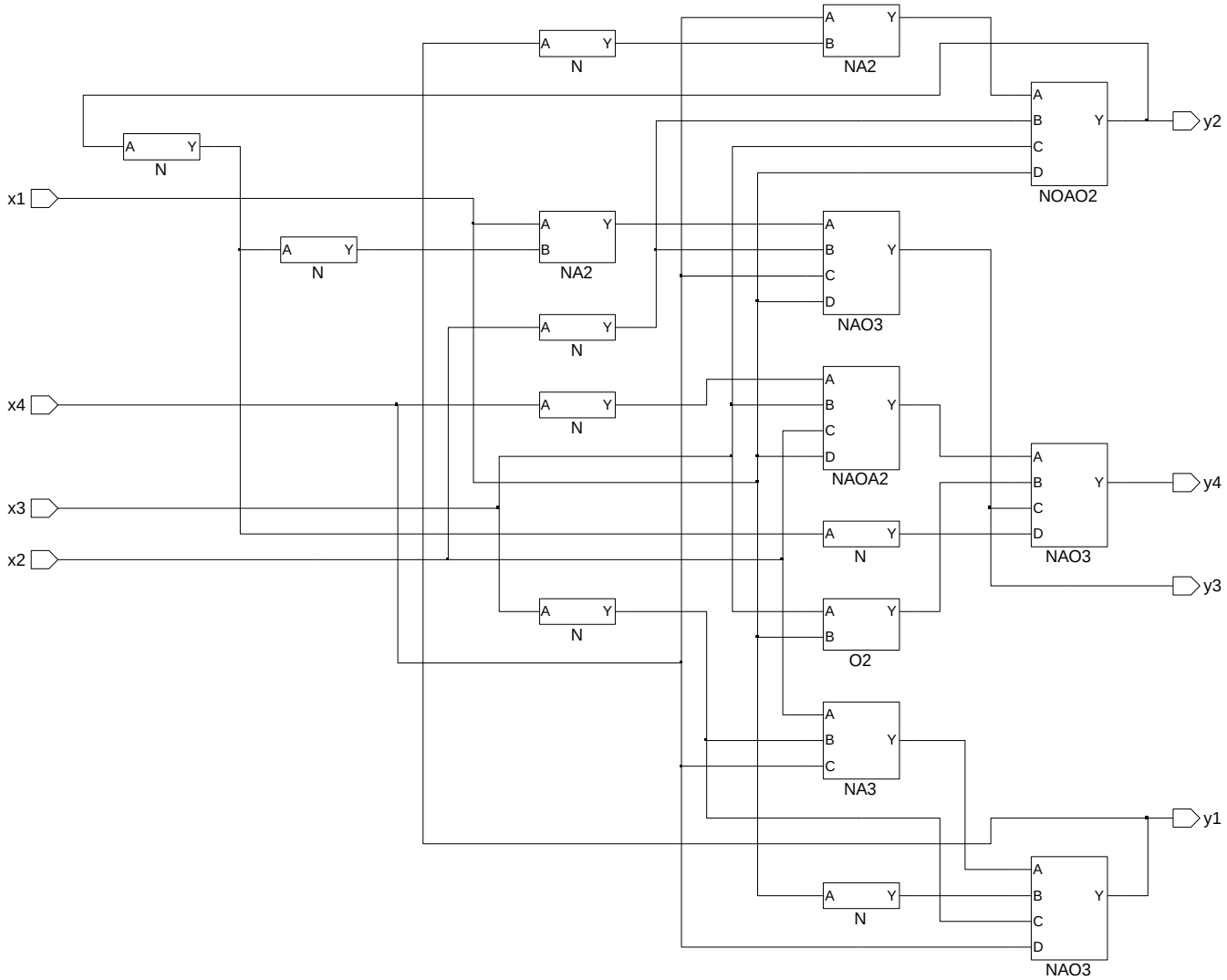

Рис. П.2. Вариант 2

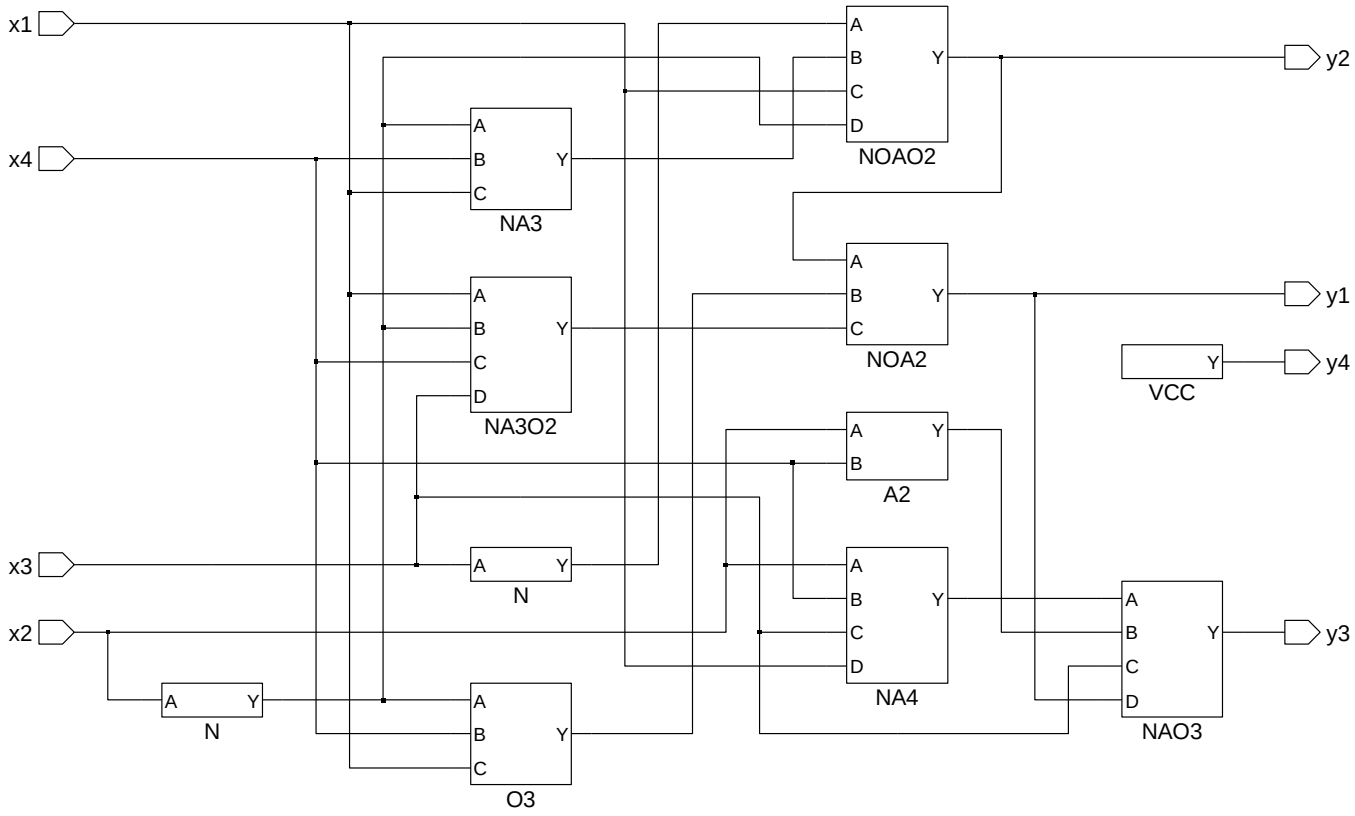

Рис. П.3. Вариант 3

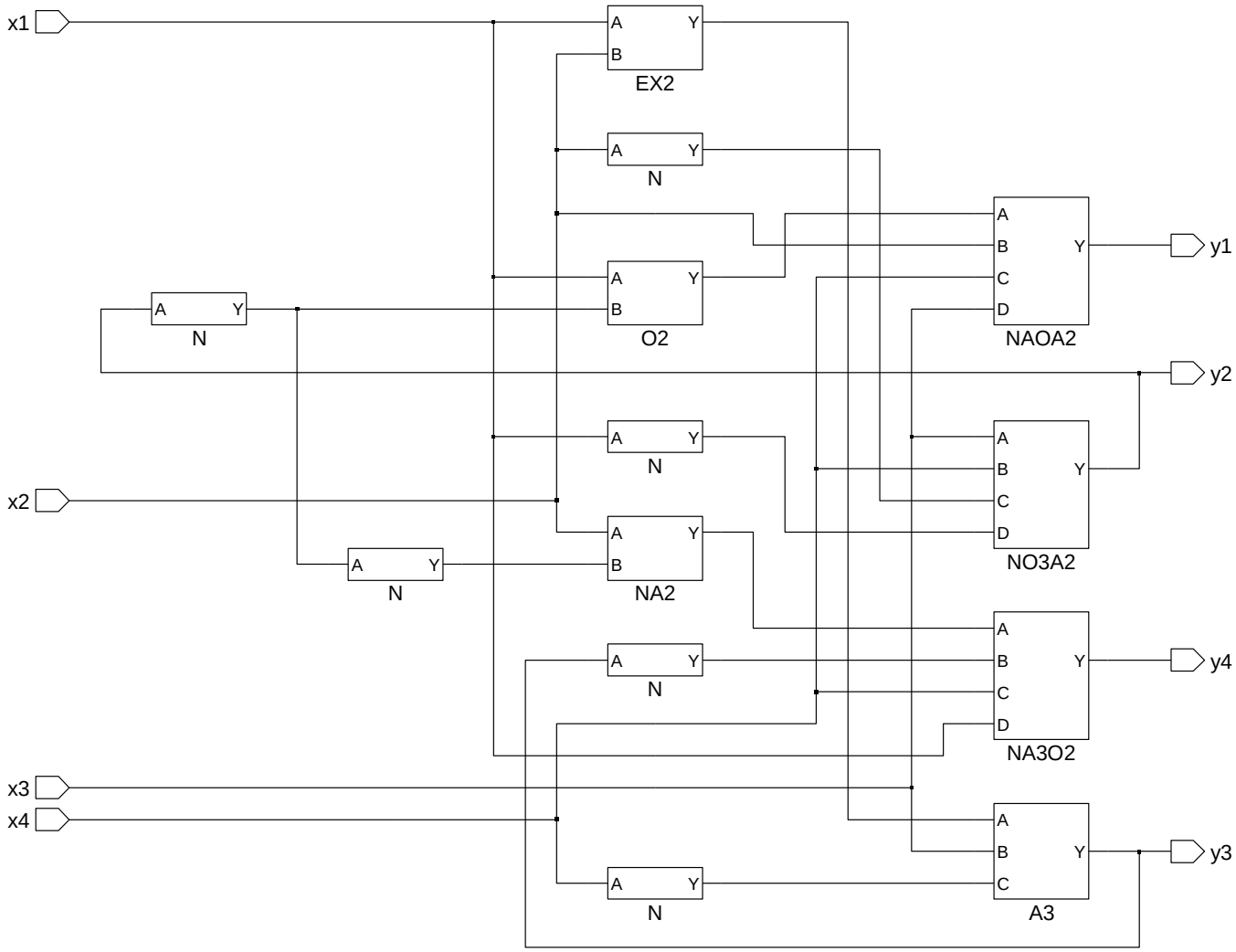

Рис. П.4. Вариант 4

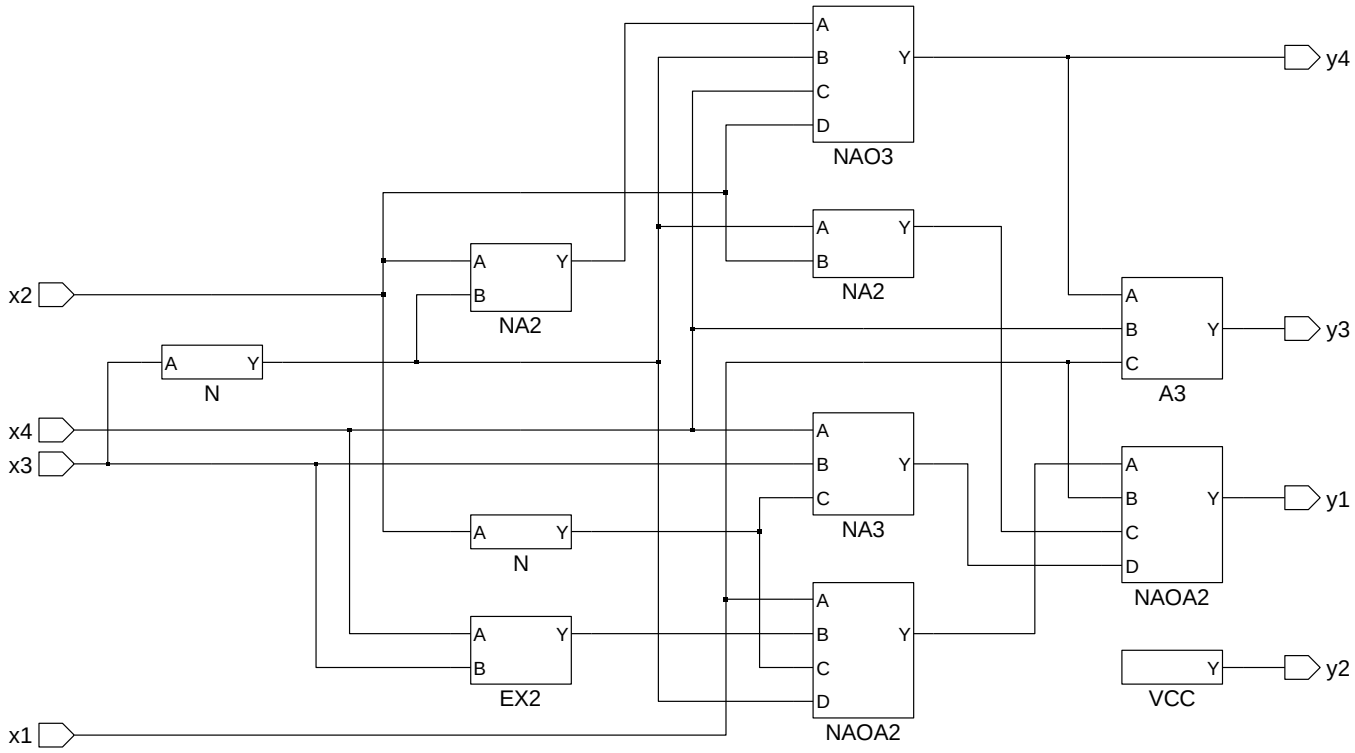

Рис. П.5. Вариант 5

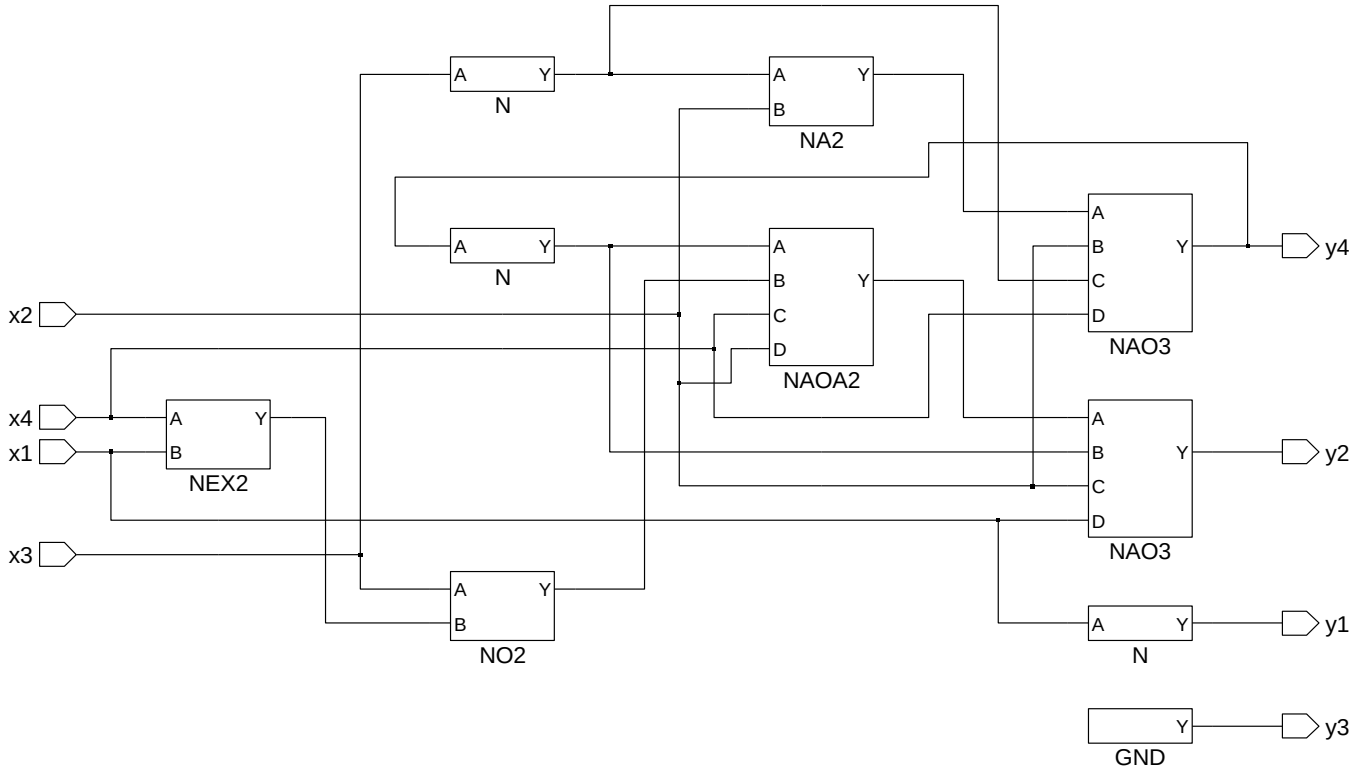

Рис. П.6. Вариант 6

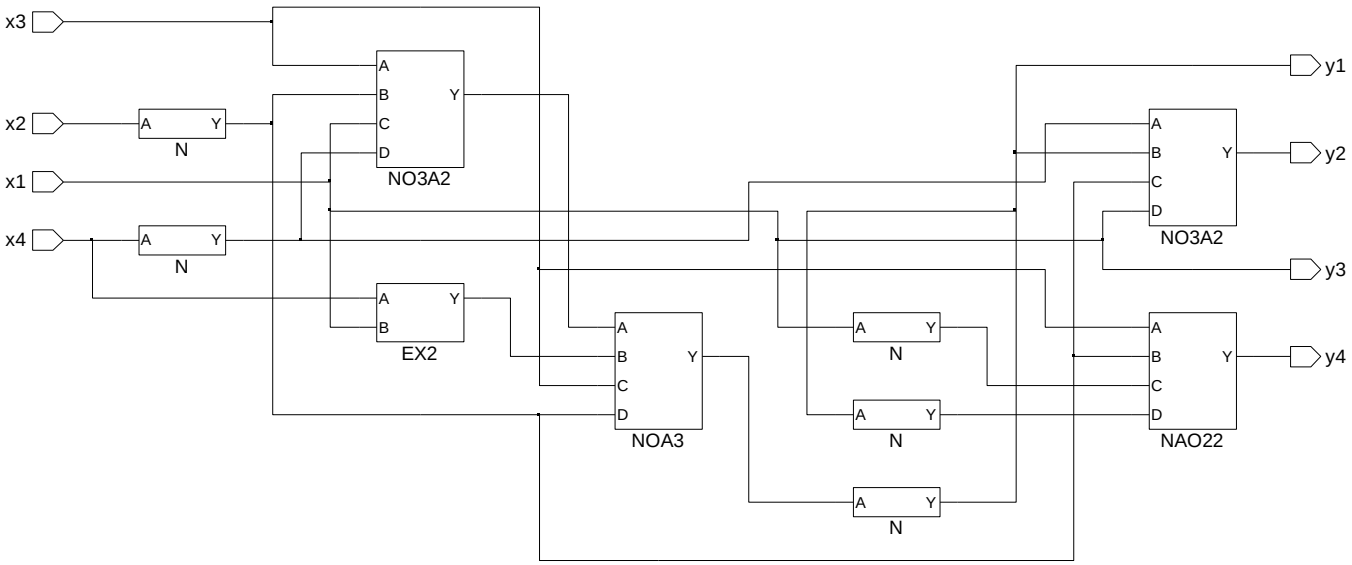

Рис. П.7. Вариант 7

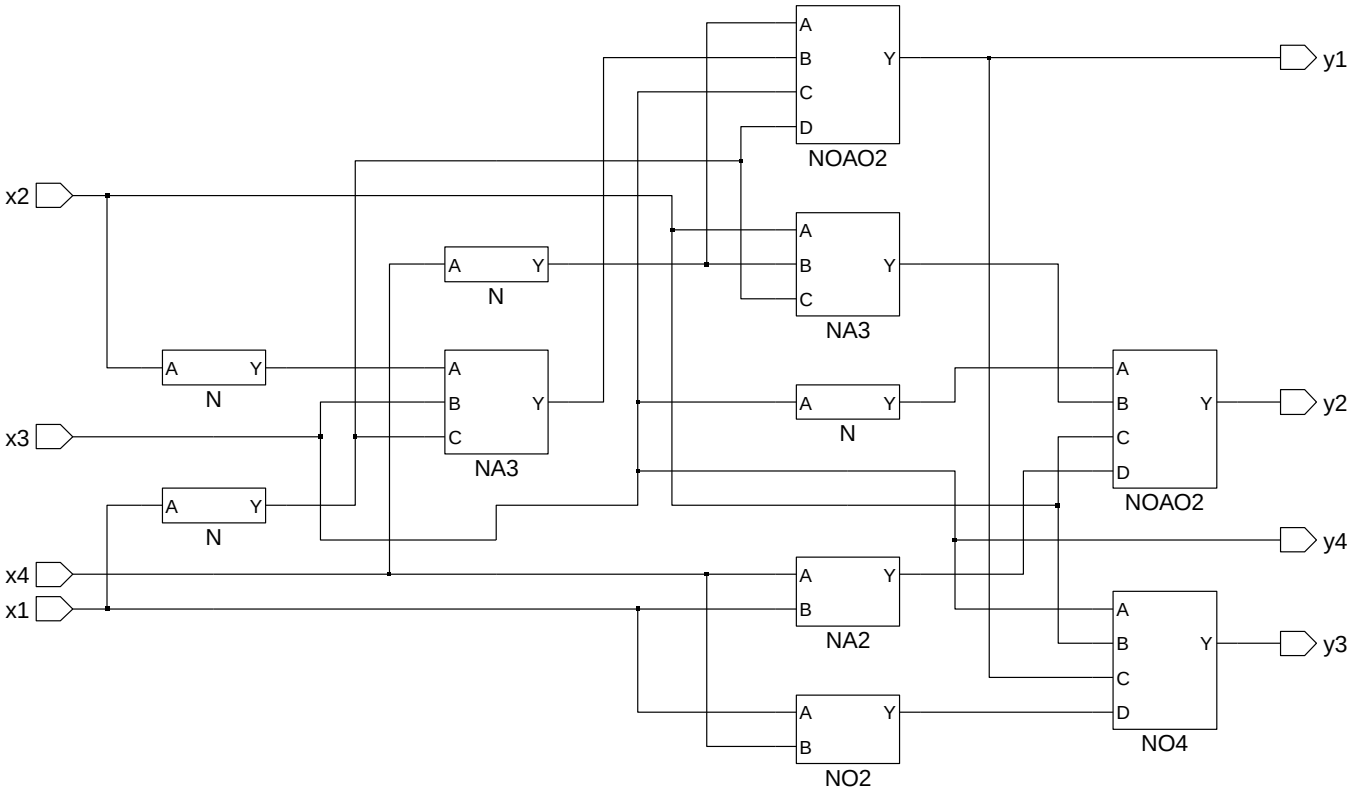

Рис. П.8. Вариант 8

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Харрис, С. Цифровая схемотехника и архитектура компьютера: RISC-V / С. Харрис, Д. Харрис. – М. : ДМК Пресс, 2021. – 810 с.
2. Наваби, З. Проектирование встраиваемых систем на ПЛИС / З. Наваби. – М. : ДМК Пресс, 2016. – 464 с.
3. Бруно, Ф. Программирование FPGA для начинающих / Ф. Бруно – М. : ДМК Пресс, 2022. – 304 с.
4. Байрак, С. А. Автоматизация проектирования ЭВМ : лаборатор. практикум для студентов специальности 1-40 02 01 «Вычисл. машины, системы и сети» всех форм обучения / С. А. Байрак, М. М. Татур. – Минск : БГУИР, 2008. – 44 с.
5. Петровский, А. А. Проектирование ЭВС с динамически реконфигурируемой архитектурой : метод. пособие для студентов специальности 1-40 02 02 «Электрон. вычисл. средства» дневной формы обучения / А. А. Петровский, М. И. Вашкевич, М. М. Родионов. – Минск : БГУИР, 2011. – 40 с.
6. Проектирование аппаратно-программных вычислительных средств: метод. пособие для студентов специальности «Программное обеспечение информ. технологий» дневной и дистанц. форм обучения / А. А. Иванюк [и др.]. – Минск : БГУИР, 2005. – 59 с.
7. Аношенко, А. Е. Лабораторный практикум по курсу «Автоматизация конструкторско-технологического проектирования электронных вычислительных средств» для студентов специальности «Проектирование и технология электронных вычислительных средств» / А. Е. Аношенко. – Минск : БГУИР, 2002. – 29 с.
8. Стешенко, В. Б. Основы HDL Verilog как средства проектирования цифровых устройств : учеб. пособие / В. Б. Стешенко, Т. В. Попова, Д. Б. Малашевич ; под ред. А. И. Сухопарова. – М. : МИЭТ, 2006. – 136 с.
9. Тарасов, И. Е. ПЛИС Xilinx. Языки описания аппаратуры VHDL и Verilog, САПР, приемы проектирования / И. Е. Тарасов. – М. : Горячая линия – Телеком, 2024. – 538 с.
10. Соловьёв, В. В. Язык SystemVerilog для синтеза / В. В. Соловьёв. – М. : Горячая линия – Телеком, 2024. – 440 с.
11. Бибило, П. Н. Моделирование и верификация цифровых систем на языке VHDL / П. Н. Бибило, Н. А. Авдеев – М. : ЛЕНАНД, 2017. – 344 с.
12. Поляков, А. К. Языки VHDL и Verilog в проектировании и цифровой аппаратуры / А. К. Поляков. – М. : СОЛОН-Пресс, 2003. – 320 с.
13. Угрюмов, Е. П. Цифровая схемотехника : учеб. пособие для вузов / Е. П. Угрюмов. – 3-е изд., перераб. и доп. – СПб. : БХВ-Петербург, 2010. – 816 с.
14. Бибило, П. Н. Задачи по проектированию логических схем с использованием языка VHDL : учеб. пособие / П. Н. Бибило. – М. : ЛКИ, 2010. – 328 с.
15. Бибило, П. Н. Основы языка VHDL / П. Н. Бибило. – 3-е изд., доп. – М. : ЛКИ, 2010. – 328 с.

16. Петровский, Ал. А. Быстрое прототипирование систем мультимедиа от прототипа // Ал. А. Петровский, А. В. Станкевич, А. А. Петровский. – Минск : Бестпринт, 2011. – 412 с.

17. Суворова, Е. А. Проектирование цифровых систем на VHDL / Е. А. Суворова, Ю. Е. Шейнин. – СПб. : БХВ-Петербург, 2003. – 576 с.

18. FPGA Tutorial [Электронный ресурс]. – Режим доступа: https://fpgatutorial.com. – Дата доступа: 03.07.2024.

19. Mano, M. Digital Design. Forth edition / M. Mano, M. Ciletti. – New Jersey : Prentice Hall, 2006. – 624 p.

Св. план 2024, поз. 33

Учебное издание

Вашкевич Максим Иосифович
Пригодич Виктор Николаевич

ПРОЕКТИРОВАНИЕ ЦИФРОВЫХ УСТРОЙСТВ НА ЯЗЫКАХ VHDL И SYSTEMVERILOG. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

ПОСОБИЕ

Редактор *А. Ю. Шурко*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *А. А. Лущикова*

Подписано в печать 19.09.2025. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 5,46. Уч.-изд. л. 5,5. Тираж 60 экз. Заказ 25.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск