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

Кафедра электронной техники и технологии

ПРОЕКТИРОВАНИЕ НА ОСНОВЕ МИКРОКОНТРОЛЛЕРОВ

Лабораторный практикум
для студентов специальности
«Медицинская электроника»
дневной и заочной форм обучения

В 2-х частях

Часть 1

Минск БГУИР 2012

УДК 004.3'12(076.5)
ББК 32.973.26-02я73
П79

А в т о р ы:
С. В. Кракасевич, М. В. Давыдов, А. В. Смирнов,
А. С. Терех, В. Л. Смирнов

Р е ц е н з е н т:
доцент кафедры сетей и устройств телекоммуникаций
учреждения образования «Белорусский государственный университет
информатики и радиоэлектроники»,
кандидат технических наук А. А. Борискевич

Проектирование на основе микроконтроллеров. : лаб. практикум
П79 для студ. спец. «Медицинская электроника» днев. и заоч. форм обуч.
В 2 ч. Ч.1 / С. В. Кракасевич [и др.]. – Минск : БГУИР, 2012. – 51 с. : ил.
ISBN 978-985-488-721-0 (ч.1).

В практикуме приводится описание четырех лабораторных, излагающих основы работы с языком VHDL при проектировании электронной аппаратуры.

Предназначен для закрепления и углубления теоретических знаний, совершенствования практических навыков в области использования информационных технологий при автоматизированном проектировании электронной аппаратуры на ПЭВМ.

Может быть использован студентами, обучающимися по специальностям «Медицинская электроника», «Проектирование и производство РЭС», «Электронно-оптические системы и технологии». Издание начинает серию лабораторных практикумов для данных специальностей.

УДК 004.3'12(076.5)
ББК 32.973.26-02я73

ISBN 978-985-488-721-0 (ч.1)
ISBN 978-985-488-735-7

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Программируемые логические интегральные схемы (ПЛИС) представляют собой одно из самых интересных и быстроразвивающихся направлений современной цифровой микроэлектроники. За последнее десятилетие наблюдался бурный рост рынка этих устройств и существенное улучшение их характеристик. Проектирование и выпуск небольшой партии уникальных цифровых устройств стали возможны в условиях проектно-конструкторских подразделений промышленных предприятий, в исследовательских и учебных лабораториях и даже в условиях домашних радиолюбительских рабочих мест. Промышленно выпускаемые «заготовки» программируемых микросхем с электрическим программированием и автоматизированным процессом перевода схемы пользователя в последовательность импульсов программирования делают проектирование новых цифровых устройств сравнимым с разработкой программного обеспечения.

Курс лабораторных работ по дисциплине «Проектирование на основе микроконтроллеров», посвященный проектированию цифровых систем на базе ПЛИС на языке *VHDL*, призван познакомить студентов с современной методикой проектирования микроэлектронных систем. Выполнение данных лабораторных работ поможет в получении необходимой базовой подготовки для дальнейшей работы в качестве инженера по разработке и обслуживанию современных микроэлектронных систем медицинского назначения.

В.1 Общие теоретические сведения по проектированию схем на VHDL

Основная тенденция при создании языка VHDL заключалась в воплощении принципов структурного программирования. Ключевая идея состояла в том, чтобы задать интерфейс разрабатываемого схемного модуля, а его внутреннее устройство скрыть. Таким образом, объект (*entity*) в языке VHDL – это объявление входов и выходов модуля, а архитектура (*architecture*) – описание внутренней структуры модуля.

В текстовом файле на языке VHDL *объявление объекта (entity declaration)* и *определение архитектуры (architecture definition)* разделены. В листинге 1 в качестве примера приведена очень простая программа на языке VHDL для 2-входового вентиля «запрета». В больших проектах объекты и архитектуры иногда бывают помещены в отдельные файлы, связь между которыми компилятор обнаруживает по их объявленным именам.

Листинг 1 – Программа на языке VHDL для вентиля «запрета»

```
entity Inhibit is                       -- also known as 'BUT-NOT'
port (X,Y: in BIT;                      -- as in 'X but not Y'
        Z: out BIT);
end Inhibit;

architecture Inhibit_arch of Inhibit is
begin
Z <= '1' when X='1' and Y='0' else '0';
end Inhibit_arch;
```

Как и в других языках программирования, в языке VHDL пробелы и переходы с одной строки на другую в общем случае игнорируются, и для удобства чтения их можно вставлять как угодно. *Комментарии (comments)* начинаются с двух дефисов (--) и заканчиваются концом строки.

В языке VHDL определено много специальных строк символов, называемых *зарезервированными словами (reserved words)* или *ключевыми словами (keywords).* В приведенном примере имеется несколько ключевых слов: entity, port, is, in, out, end, architecture, begin, when, else и not. Определяемые пользователем *идентификаторы (identifiers)* начинаются с буквы и содержат буквы, цифры и подчеркивания. (Символ подчеркивания не может следовать за другим символом подчеркивания и не может быть последним символом идентификатора.) В данном примере идентификаторами являются Inhibit, X, Y, BIT, Z и Inhibit_arch. «BIT» – это встроенный идентификатор предопределенного типа. Зарезервированные слова и идентификаторы нечувствительны к регистру.

В листинге 2 представлен синтаксис объявления объекта. Целью объявления объекта помимо присвоения объекту имени является определение сигналов внешнего интерфейса или *портов (ports)* в части объявления объекта, которая называется *объявлением портов (port declaration).* Кроме ключевых слов entity, is, port и end *объявление* объекта содержит следующие элементы:

entity-name – выбираемое пользователем имя объекта;

signal-names – список выбираемых пользователем имен сигналов внешнего интерфейса, состоящий из одного имени или из большего числа имен, разделенных запятой;

mode – одно из четырех зарезервированных слов, определяющих направление передачи сигнала:

in – сигнал на входе объекта;

out – сигнал на выходе объекта; заметьте, что значение такого сигнала нельзя «прочитать» внутри структуры объекта; он доступен только объектам, использующим данный объект;

buffer – сигнал на выходе объекта, такой, что его значение можно читать также внутри структуры данного объекта;

inout – сигнал, который может быть входным или выходным для данного объекта; обычно этот режим используется применительно к входам/выходам ПЛУ с тремя состояниями;

signal-type – встроенный или определенный пользователем тип сигнала; в следующих разделах мы будем много говорить об этом.

Листинг 2 – Синтаксис объявления объекта на языке VHDL

```
entity entity-name is
  port(signal-names  :  mode signal-type;
       signal-names  :  mode signal-type;
       ...
       signal-names :  mode signal-type);
end entity-name;
```

Порты объекта, а также направление передачи и типы сигналов – это все, что видят другие модули, использующие данный модуль. Внутренняя работа объекта задается его *определением архитектуры (architecture definition),* синтаксис которого в общем случае имеет вид, указанный в листинге 3 *Имя объекта (entity-name)* в этом определении должно быть таким же, какое раньше было присвоено объекту в объявлении объекта. *Имя архитектуры (architecture-name)* – это выбираемый пользователем идентификатор, обычно так или иначе связанный с именем объекта; при желании имя архитектуры может быть тем же самым, что и имя объекта.

Листинг 3 – Синтаксис определения архитекрутуры на языке VHDL

```
architecture architecture-name of entity-name is
type declarations
signal declarations
constant declarations
function definitions
procedure definitions
component declarations
begin
concurrent-statement
...
concurrent-statement
end architecture-name;
```

Сигналы внешнего интерфейса архитектуры (порты) наследуются от той части объявления соответствующего объекта, где объявляются порты. У архитектуры могут быть также сигналы и другие объявления, являющиеся для нее локальными, подобно тому, как это имеет место в других языках высокого уровня. В отдельном «пакете», используемом несколькими объектами, можно сделать объявления, общие для этих объектов, о чем будет сказано позднее.

Объявления в листинге 3 могут располагаться в произвольном порядке. Начать легче всего с *объявления сигнала (signal declaration),* которое сообщает ту же самую информацию о сигнале, какую содержит объявление порта, за исключением того, что вид сигнала не задается:

```
signal signal-names : signal-type;
```

В архитектуре может быть объявлено любое число сигналов, начиная с нуля, и они приблизительно соответствуют поименованным соединениям в принципиальной схеме. Их можно считывать и записывать внутри определения архитектуры и, подобно другим локальным элементам, на них можно ссылаться только в пределах данного определении архитектуры.

Переменные (variables) в языке VHDL похожи на сигналы, за исключением того, что, как правило, они не имеют никакого физического смысла в схеме. Действительно, обратите внимание, что, согласно листингу 3, в определении архитектуры не предусмотрено «объявление переменных». Переменные используются в функциях, процедурах и процессах языка VHDL. Каждый из этих элементов программы мы рассмотрим позднее. Вот у них внутри имеются *объявления переменных (variable definitions),* и эти объявления в точности подобны

объявлениям сигналов, за исключением того, что употребляется ключевое слово variable:

```
variable  variable-names  :  variable-type;
```

Каждому сигналу, переменной и константе в программе на языке VHDL необходимо поставить в соответствие *тип (type).* Типом определяется множество или диапазон значений, которые может принимать данный элемент, и обычно имеется набор операторов (таких как сложение, логическое И и т. д.), связываемых с данным типом.

В языке VHDL есть всего лишь несколько *предопределенных типов (predefined types);* они перечислены в таблице В.1. В дальнейшем будут использованы только следующие предопределенные типы: integer, character и boolean. Вы можете подумать, что при цифровом проектировании большую роль должны играть имена «bit» и «bit_vector», но оказывается, что более полезны определяемые пользователем варианты этих типов, как это вскоре будет объяснено.

Таблица В.1 – Предопределенные типы языка VHDL

bit	character	severity_level
bit_vector	integer	string
boolean	real	time

Встроенные операторы для типов integer и boolean приведены в таблице В.2.

Таблица В.2 – Предопределенные операторы для типов integer и boolean в языке VHDL

Операторы для типа *integer*		Операторы для типа *boolean*	
+	Сложение	and	И
-	Вычитание	or	ИЛИ
*	Умножение	nand	И-НЕ
/	Деление	nor	ИЛИ-НЕ
mod	Деление по модулю	xor	ИСКЛЮЧАЮЩЕЕ ИЛИ
rem	Остаток от деления по модулую	xnor	ИСКЛЮЧАЮЩЕЕ ИЛИ-НЕ
abs	Абсолютное значение	not	Дополнение (инверсия)
**	Возведение в степень		

Чаще всего в типичных программах на языке VHDL используются *определяемые пользователем типы (user-defined types),* а из них самыми

употребительными являются *перечислимые типы (enumerated types),* которые определяются путем перечисления их значений. Предопределяемые типы boolean и character – это перечислимые типы. Формат объявления типа в случае перечислимого типа указан в первой строке листинга 4. Здесь *value-list* представляет собой список (перечисление) всех возможных значений этого типа, разделяемых запятыми. Значениями могут быть определяемые пользователем идентификаторы или символы (где под «символом» понимается символ ISO, заключенный в одинарные кавычки). Идентификаторы чаще всего применяются для обозначения альтернатив или состояний конечного автомата, например:

```
type traffic_light_state is (reset, stop, wait, go);
```

Символы используются в очень важном случае стандартного определяемого пользователем логического типа std_logic. Этот тип включает не только «0» и «1», но также и семь других значений, которые оказываются полезными при моделировании логического сигнала (бита) в реальной логической схеме.

Листинг 4 – Синтаксис объявления типов и констант в языке VHDL

```
type type-name is (value-list);
subtype subtype-name is type-name start to end;
subtype subtype-name is type-name start downto end;
constant constant-name: type-name := value;
```

Язык VHDL позволяет пользователю создавать также *подтипы (subtypes)* согласно синтаксису, указанному в листинге 4. Значения подтипа должны быть слитным подмножеством значений, предусмотренных основным типом, начиная со *start* и кончая *end.* Для перечислимого типа «слитность» означает расположение на соседних позициях в исходном списке значений *value-list.* Вот несколько примеров определения подтипов:

```
subtype twoval_logic is std_logic range '0' to '1';
subtype fourval_logic is std_logic range 'X' to 'Z';
subtype negint is integer range -2147483647 to -1;
subtype bitnum is integer range 31 downto 0;
```

Заметьте, что порядок следования значений в указываемом диапазоне может быть в сторону возрастания или в сторону убывания в зависимости от того, какое из ключевых слов *to* или *downto* употреблено. Из-за некоторых особенностей подтипов это различие может быть существенным. В языке VHDL есть два предопределенных подтипа integer:

subtype natural is integer range 0 to *highest-integer;*

subtype positive is integer range 1 to *highest-integer;*

Константы (constants) способствуют удобству чтения программ, возможности их поддержания и сопровождения, а также переносу на какой-либо другой язык. Синтаксис *объявления констант (constant declaration)* в языке VHDL указан в последней строке листинга 4; его можно проиллюстрировать следующими примерами:

```
constant BUS_SIZE: integer := 32;    -- width of component
constant MSB: integer  := BUS_SIZE-1; -- bit number of MSB
constant Z: character  :=  'Z';       -- synonym for Hi-Z value
```

Другую очень важную группу определяемых пользователем типов образуют

типы массивов (array types). Как и в других языках, в языке *VHDL массив (array)* по определению – это упорядоченный набор элементов одного и того же типа, отдельные компоненты которого выбираются с помощью *индекса массива (array index).* Возможны несколько вариантов синтаксиса объявления массива в языке VHDL; они представлены в листинге 5. В первых двух вариантах *start* и *end* являются целыми числами, которыми задается возможный диапазон изменения индекса массива и, следовательно, полное число элементов массива. В последних трех вариантах диапазоном изменения индекса массива являются все значения указанного типа *(range-type)* или подмножество этих значений.

Листинг 5 – Синтаксис объявления массивов в языке VHDL

```
type type-name is array (start to end)  of element-type;
type type-name is array (start downto end)  of element-type;
type type-name is array (range-type)  of element-type;
type type-name is array (range-type range start to end) of element-type;
type type-name is array (range-type range start downto end) of element-type;
```

В листинге 6 приведены примеры объявления массивов. Первые два примера совсем обычны и демонстрируют задание диапазона изменения индекса в сторону возрастания и в сторону убывания. Следующий пример показывает, как можно воспользоваться константой WORDLEN при объявлении массива; отсюда видно также, что границу диапазона можно задать простым выражением. Из третьего примера следует, что сам элемент массива может быть массивом; таким образом создается двумерный массив. Последний пример показывает, что множество возможных значений элементов массива можно задать, указав перечислимый тип (или подтип); в этом примере массив состоит из четырех элементов согласно данному нами чуть раньше определению типа traffic_light_state. Элементы массива считаются упорядоченными слева направо в том же направлении, в каком индекс пробегает свои значения. Таким образом, индексы самых левых элементов массивов типов monthly_count, byte, word, reg_file и statecount в листинге 6 равны 1, 7, 31, 1 и reset соответственно.

Листинг 6 – Примеры объявления массивов в языке VHDL

```
type monthly_count is array  (1 to 12) of integer;
type byte is array  (7 downto 0)   of STD_LOGIC;
constant WORD_LEN:   integer   := 32;
type word is array  (WORD_LEN-1 downto 0)  of STD_LOGIC;
constant NUM_REGS:   integer   := 8;
type reg_file is array  (1 to NUM_REGS)  of word;
type statecount is array  (traffic_light_state)  of integer;
```

Обращение к отдельным элементам массивов в операторах программы на языке VHDL осуществляется путем указания имени массива и индекса элемента в круглых скобках. Если, например, M, B, W, R и S – сигналы или переменные тех пяти типов массивов, которые приведены в листинге 6, то любая из записей M (11), B (5), W (WORD_LEN-5), R(0,0), R(0) и S(reset) является правильным указанием элемента.

Можно также указывать подмножество непосредственно следующих один за другим элементов массива или, как говорят, *вырезку из массива (array slice),* задавая начальный и конечный индексы подмножества; например: M (6 to 9), B(3 downto 0), W(15 downto 8), R(0,7 downto 0), R(1 to 2), S(stop to go). Заметьте, что направление изменения индекса в вырезке должно быть таким же, как у исходного массива.

Наконец, массивы или элементы массивов можно объединять с помощью *оператора конкатенации* & *(concatenation operator),* который соединяет массивы и элементы в том порядке, в каком они записаны слева направо. Например, запись '0' & '1' & "1Z" эквивалентна строке "011Z", а выражение B (6 downto 0) & B (7) представляет собой циклический сдвиг 8-разрядного массива B на 1 разряд влево.

Самым важным типом массивов в типичной программе на языке VHDL является определяемый пользователем в соответствии со стандартом IEEE 1164 логический тип std_logic_vector, которым задается упорядоченный набор элементов типа std_logic. Определение этого типа имеет вид

```
type STD_LOGIC_VECTOR is array (natural range <>) of STD_LOGIC;
```

Это пример *типа массива без ограничений (unconstrained array type)*: диапазон возможных значений индекса массива не задан, за исключением того, что он должен быть подмножеством определенного типа, в данном случае – типа natural. Эта особенность языка VHDL позволяет записывать архитектуры, функции и другие элементы программ в более общем виде, до некоторой степени независимо от размеров массивов и диапазонов возможных значений индексов. Действительный диапазон значений индекса определяется в тот момент, когда сигналу или переменной ставится в соответствие этот тип.

B.2 Функции и процедуры

Подобно функции в любом языке программирования высокого уровня, *функция (function)* в языке VHDL получает ряд *аргументов (arguments)* и возвращает *результат (result).* При определении функции на языке VHDL и при ее вызове каждый из аргументов и результат имеют предустановленный тип.

Синтаксис *определения функции (function definition)* приведен в листинге 7. За присвоением функции определенного имени следует список *формальных параметров (formal parameters),* который используется в теле функции; число параметров может быть любым, начиная с нуля. При вызове функции формальные параметры в обращении к ней замещаются *действительными параметрами (actual parameters).* В соответствии со строгим следованием типам в языке VHDL действительные параметры должны быть того же типа или подтипа, что и формальные параметры. Когда функция вызывается из архитектуры, на место ее вызова возвращается значение, тип которого указывается посредством *return-type.*

Листинг 7 – Синтаксис определения функции в языке VHDL

```
function function-name (
```

```
signal-names : signal-type;
...
signal-names : signal-type)
return return-type is
type declarations
constant declarations
variable declarations
function definitions
procedure definitions
begin
sequential-statement
...
sequential-statement
end function-name;
```

Как видно из листинга 7, внутри функции можно определить ее собственные локальные типы, константы, переменные и вложенные функции и процедуры, Между ключевыми словами begin и end располагается ряд «последовательных операторов», которые исполняются при вызове функции. Последовательные операторы и их синтаксис будут предметом более подробного разбора позднее, но вам должны быть понятны приводимые здесь примеры с учетом вашего предыдущего опыта программирования.

Архитектуру «вентиля запрета» на языке VHDL, приведенную в лист. 1, можно видоизменить, используя функцию, как показано в листинге 8. В определении функции момент возврата к месту вызова указывается ключевым словом *return,* за которым следует выражение, значение которого и будет возвращено. Тип результата вычисления этого выражения должен быть согласован со значением *return-type* в объявлении функции.

Листинг 8 – Программа для «функции запрета» на языке VHDL

```
architecture Inhibit_archf of Inhibit is
function ButNot (A, B: bit) return bit is
begin
if B = '0' then return A;
else return '0';
end if;
end ButNot;
begin
Z <= ButNot(X,Y);
end Inhibit_archf;
```

В.3 Библиотеки и пакеты

VHDL-библиотека (library) – это место, где компилятор VHDL хранит информацию об отдельном варианте проекта, включая промежуточные файлы, используемые при анализе, моделировании и синтезе в рамках данной разработки. Место библиотеки в файловой системе компьютера зависит от реализации. Для очередного VHDL-проекта компилятор автоматически создает библиотеку под именем «work» и использует ее.

Даже в малых проектах может использоваться библиотека, которая содержит, например, стандартные определения IEEE. Разработчик может присвоить имя такой библиотеке с помощью *предложения library (library clause)* в начале

соответствующего файла. Например, библиотеку IEEE можно задать фразой

```
library  ieee;
```

Присвоение имен библиотекам проекта обеспечивает доступ к любым ранее проанализированным и запомненным объектам и архитектурам, но не к определениям типов и тому подобному. Эту функцию выполняют «пакеты» и «предложения use».

```
use ieee.std_logic_1164.all;
```

Здесь "ieee" – это имя библиотеки, ранее введенное предложением library. В этой библиотеке файл с именем "std_logic_1164" содержит желаемые определения. Приставка "all" велит компилятору использовать все определения этого файла. Вместо "all" можно написать имя какого-то одного элемента, когда необходимо использовать только его определение, например:

```
use ieee.std_logic_1164.std_ulogic;
```

Эта фраза обеспечит доступ только к определению типа std_logic, приведенному в листинге 11, без учета всех других родственных типов и функций. Однако, записывая подряд несколько предложений "use", можно добавить использование и других определений.

В.4 Элементы структурного проектирования

Тело архитектуры представляет собой ряд *параллельных операторов (concurrent statements).* В языке VHDL каждый параллельный оператор исполняется одновременно с другими параллельными операторами в теле данной архитектуры.

В языке VHDL есть несколько параллельных операторов, а также механизм связывания в один узел набора последовательных операторов, с тем чтобы они исполнялись, как один параллельный оператор. Различное использование параллельных операторов привело к возникновению трех стилей проектирования и описания схем, слегка отличающихся один от другого.

Самым главным параллельным оператором в языке VHDL является *оператор component (component statement),* синтаксис которого указан в листинге 9. Здесь *component-name* – имя определенного ранее объекта, который должен быть использован или *подвергнут обработке (instantiate)* в теле данной архитектуры. Для каждого оператора component с обращением к названному объекту создается одна копия этого объекта; каждая копия должна иметь уникальную метку *label.*

Листинг 9 – Синтаксис оператора component в языке VHDL

```
label:component-name port map(signal1, signnal2,   ...,  signaln);
label:component-name port map (port1=>signal1, port2=>signal2, ...,
portn=>stgnaln);
```

Ключевое слово port map вводит список, посредством которого портам названного объекта ставятся в соответствие сигналы данной архитектуры. Список может быть представлен одним из двух различных способов. Первый из них является позиционным: как и в обычных языках программирования, сигналы,

упоминаемые в списке, связываются с портами объекта в том же самом порядке, в каком порты перечислены в определении объекта. Второй способ записи – явный: каждый порт объекта связывается с сигналом посредством оператора "=>", и эти соответствия могут следовать в любом порядке.

До того как компонент будет подвергнут обработке внутри архитектуры, он должен быть декларирован *объявлением компонента (component declaration)* в определении архитектуры (см. листинг 3). Как видно из листинга 10, объявление компонента является по существу таким же, что и часть объявления соответствующего объекта, где объявляются порты: приводятся имя, режим и тип каждого порта.

Листинг 10 – Синтаксис объявления компонента в языке VHDL

```
component component-name
port  (signal-names : mode signal-type;
  ...
  signal-names : mode signal-type);
end component;
```

Используемые в архитектуре компоненты могут быть либо ранее определенными элементами данного проекта, либо библиотечными элементами. Листинг 11 представляет собой пример VHDL-обьекта и его архитектуры, в которой используются компоненты «устройства для обнаружения простых чисел». В объявлении объекта названы входы схемы и ее выход. В части архитектуры, отведенной под объявления, присваиваются имена всем сигналам и компонентам, которые используются внутри данной архитектуры (INV, AND2, AND3 OR4 являются предопределенными компонентами).

VHDL-архитектуру, в которой используются компоненты, часто называют структурным описанием (structural description) или структурной моделью *(structural design)*, поскольку ею задается реализующая данный объект точная конфигурация соединений, по которым сигналы передаются от одного элемента к другому. В этом отношении ясное структурное описание эквивалентно схеме устройства или списку соединений в нем.

Листинг 11 – Структурная VHDL-программа для устройства, обнаруживающего простые числа

```
library IEEE;
use IEEE.std_logic_1164.all;
entity prime is
port ( N: in STD_LOGIC_VECTOR (3 downto 0);
 F:  out STD_LOGIC );
end prime;

architecture prime1_arch of prime is
signal N3_L, N2_L,  N1_L:   STD_LOGIC;
signal N3L_N0, N3L_N2L_N1,  N2L_N1_N0,  N2_N1L_N0:  STD_LOGIC;
component INV port  ( I:  in STD_LOGIC;
                      O:  out STD_LOGIC);
end component;
component AND2 port ( I0,I1:  in STD_LOGIC;
```

```
                        O:  out STD_LOGIC);
end component;
component AND3 port  (I0,I1,I2:   in STD_LOGIC; O: out STD_LOGIC);
end component;
component OR4 port (I0,I1,I2,I3: in STD_LOGIC; O: out STD_LOGIC);
end component;
begin
Ul:  INV port map (N(3),  N3_L);
U2:  INV port map (N(2),  N2_L);
U3:  INV port map (N(l), N1_L);
U4:  AND2 port map (N3_L,  N(0),  N3L_N0);
U5:  AND3 port map (N3_L,  N2_L,  N(l),  N3L_N2L_N1);
U6:  AND3 port map (N2_L,  N(l),  N(0),  N2L_N1_N0);
U7:  AND3 port map (N(2),  Nl_L,  N(0),  N2_N1L_N0);
U8:  OR4 port map (N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0, F);
end prime1_arch;
```

В некоторых приложениях бывает необходимо создать несколько копий определенного блока внутри архитектуры. Язык VHDL имеет *оператор generate (generatestatement)*, который позволяет создавать такие повторяющиеся блоки посредством своего рода цикла «for» без необходимости выписывать все копии по отдельности.

Синтаксис простого итеративного цикла generate показан в листинге 12. Идентификатор *identifier* объявляется явно как переменная, тип который совместим с диапазоном *range*. Параллельный оператор *concurrent statement* исполняется однократно для каждого возможного значения переменной *identifier* в пределах диапазона; переменную *identifier* можно использовать внутри параллельного оператора. В листинге 14 показано, как можно построить инвертор с произвольной разрядностью.

Листинг 12 – Синтаксис цикла for-generate на языке VHDL

```
label:   for identifier in range generate
concurrent-statement
end generate;
```

Значение константы должно быть известно к моменту компиляции программы, написанной на языке VHDL. Во многих приложениях бывает полезно разработать и откомпилировать объект и его архитектуру, оставляя некоторые из его параметров не заданными, например разрядность шины. Сделать это позволяет имеющийся в языке VHDL инструмент «generic».

С помощью *объявления общности (generic declaration)* в объявлении объекта можно определить одну или большее число *настраиваемых констант (generic constant);* это необходимо сделать до объявления портов согласно синтаксису, указанному в листинге 13.

Листинг 13 – Синтаксис объявления общности в объявлении объекта

```
entity entity-name is
generic (constant-names : constant-type;
          ...
          constant-names : constant-type);
port     (signal-names  :  mode signal-type;
          ...
```

```
            signal-names  :  mode signal-type);
end entity-name;
```

Каждую поименованную константу можно использовать в определении архитектуры данного объекта, а задание ее значения откладывается до того момента, когда этот объект будет подвергаться обработке оператором component из другой архитектуры. Значения присваиваются настраиваемым константам в этом операторе component с помощью предложения *generic map* таким же способом, какой употреблен в предложении port map. В листинге 14 приведен пример, в котором одновременно используются инструмент generic и оператор generate для создания «шинного инвертора» с задаваемой пользователем разрядностью.

Листинг 14 – VHDL-объект и его архитектура для шинного инвертора с произвольной разрядностью

```
library IEEE;
use IEEE.std_logic_1164.all;

entity businv is
generic (WIDTH: positive);
port ( X: in STD_LOGIC_VECTOR (WIDTH-1 downto 0);
 Y: out STD_LOGIC_VECTOR (WIDTH-1 downto 0) );
end businv;

architecture businv_arch of businv is
component IMV port (I: in STD_LOGIC; 0: out STD_LOGIC); end component;
begin
g1:      for b in WIDTH-1 downto 0 generate
Ul:  INV port map (X(b), Y(b));
end generate;
end businv_arch;
```

B.5 Элементы потокового проектирования

Несколько дополнительных параллельных операторов языка VHDL позволяют описывать схему в терминах потока данных и выполняемых схемой операций над этими данными. Такой подход носит название *потокового описания (dataflow description)* или потокового проектирования (*dataflow design).*

В потоковых проектах используются два дополнительных параллельных оператора; они приведены в листинге 15. Чаще всего используется первый из них; он называется *параллельным сигнальным оператором присваивания (concurrent signal-assignment statement).* Его можно прочесть так: «Сигнал с именем *signal-name* принимает значение выражения *expression».* Поскольку в языке VHDL необходимо строго соблюдать типы, тип выражения *expression* должен быть совместим с типом сигнала *signal-name.* В общем случае это означает, что типы должны быть либо тождественно одинаковыми, либо тип *expression* должен являться подтипом типа *signal-name.*

Листинг 15 – Синтаксис параллельных сигнальных операторов присваивания в языке VHDL

```
signal-name <= expression;
```

```
signal-name <= expression when boolean-expression else
               ...
               expression when boolean-expression else
               expression;
```

В листинге 16 представлена архитектура объекта для устройства, обнаруживающего простые числа (см. листинг 2), записанная в потоковой форме. При таком подходе вентили и соединения между ними в явном виде не указываются; вместо этого используются встроенные операторы языка VHDL and, or и not (на самом деле для сигналов типа STD_LOGIC таких встроенных операторов нет, но они определяются и перегружаются пакетом IEEE 1164). Заметьте, что у оператора not самый высокий приоритет, так что для получения нужного результата не требуется заключать в скобки подвыражение типа "not N (3)".

Можно также воспользоваться второй, условной формой параллельного сигнального оператора присваивания (conditional signal-assignment statement) с ключевыми словами when и else, как показано в листинге 17. В этом случае в выражении boolean-expression отдельные булевы термы объединяются посредством встроенных булевых операторов языка VHDL, таких как and, or и not.

Под булевыми термами обычно понимаются булевы переменные или результаты сравнения, выполняемого с помощью *операторов отношений (relational operators)* =, /= (не равно), >, >=, < и <=.

Листинг 16 – Потоковая VHDL-архитектура для устройства, обнаруживающего простые числа

```
architecture prime2_arch of prime is
signal N3L_N0,   N3L_N2L_N1,   N2L_N1l_N0,   N2_N1L_N0:   STD_LOGIC;
begin
N3L N0     <= not N(3)                                    and      N(0);
N3L_N2L_N1 <= not N(3)  and not N(2)  and       N(1);
N2L_N1_N0  <=              not N(2)  and       N(l)    and      N(0);
N2_N1L_N0  <=                  N(2)  and   not N(l)    and      N(0);
F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
end prime2_arch;
```

Листинг 17 содержит пример использования условных параллельных операторов присваивания. Каждый бит переменной типа STD_LOGIC, например N(3), сравнивается со знаковыми литералами '1' и '0', и результат возвращается в виде значения типа boolean. Результаты сравнения объединяются в булево выражение, помещенное в каждом операторе между ключевыми словами when и else. В общем случае требуются предложения else; совокупный набор условий в каждом из операторов должен покрывать все возможные комбинации входных сигналов.

Листинг 17 – Архитектура устройства для обнаружения простых чисел, в которой использованы условные присваивания

```
architecture pnme3_arch of prime is
signal N3L_N0,  N3L_N2L_N1,  N2L_N1_N0,  N2_N1L_N0:  STD_LOGIC;
```

```
begin
N3L_N0    <= '1' when N(3)='0' and N(0)='1' else '0';
N3L_N2LN1 <= '1' when N(3)='0' and N(2)='0' and N(l)='l' else '0';
N2L_N1_N0 <= '1' when N(2)='0' and N(1)='1' and N(0)='1' else '0';
N2_N1L_N0 <= '1' when N(2)='1' and N(1)='0' and N(0)='1' else '0';
F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
end prime3_arch;
```

Параллельный оператор присваивания другого рода – это *избирательное присваивание сигналу его значения (selected signal-assignment statement),* синтаксис которого указан в листинге 18. Этот оператор вычисляет заданное выражение *expression* и присваивает сигналу с именем *signal-name* значение сигнала *signal-value,* соответствующее той из альтернатив *choices,* значение которой равно *expression.* Альтернативой в каждом предложении when может быть одиночное возможное значение *expression* или список значений, разделенных вертикальной чертой (!). Альтернативы *choices* в данном операторе должны быть взаимно исключающими и в совокупности включать все возможные случаи. В последнем предложении when можно воспользоваться ключевым словом *others* в качестве указания на все значения *expression,* которые еще не были упомянуты.

Листинг 18 – Синтаксис избирательного сигнального оператора присваивания в языке VHDL

```
with expression select
signal-name <=  signal-value when choices,
signal-value when choices,
signal-value when choices;
```

В архитектуре устройства для обнаружения простых чисел, приведенной в листинге 19, использован избирательный сигнальный оператор присваивания. Все *альтернативы,* для которых F равно T могли бы быть записаны в одном предложении when; в нашем примере они разнесены по нескольким предложениям только с учебной целью. Здесь избирательный сигнальный оператор присваивания как бы считывает запись множества включений функции F.

Листинг 19 – Архитектура устройства для обнаружения простых чисел, в которой используется избирательное присваивание сигналу его значения

```
architecture prime4_arch of prime is
begin
with N select
F <= '1'  when "0001",
     '1'  when "0010",
     '1'  when "0011"  |  "0101"  |  "0111",
     '1'  when "1011"  |  "1101",
     '0'  when others;
end prime4_arch;
```

B.6 Элементы поведенческого проектирования

Иногда параллельным оператором можно непосредственно описать требуемое поведение логической схемы. И это очень хорошо, потому что возможность *поведенческого описания (behavioral description)* и выполнение *поведенческого проекта (behavioral design)* является главным достоинством языков описания схем вообще и языка VHDL в частности. Однако для большинства поведенческих описаний нужны некоторые дополнительные элементы языка.

Ключевым поведенческим элементом языка VHDL является «процесс». *Процесс (process)* – это совокупность «последовательных» операторов (они будут описаны чуть ниже), которые выполняются одновременно с другими параллельными операторами и с другими процессами.

Оператор процесса (process statement) в языке VHDL можно использовать повсюду, где возможно употребление параллельного оператора. Оператор процесса вводится ключевым словом *process;* синтаксис этого оператора приведен в листинге 20. Оператор process пишется внутри некоторой объемлющей архитектуры, поэтому ему доступны все типы, сигналы, константы, функции и процедуры, объявленные в этой архитектуре, а также так или иначе видимые из этой архитектуры. Но можно также определять и локальные типы, переменные, константы, функции и процедуры внутри данного процесса.

Листинг 20 – Синтаксис оператора process в языке VHDL

```
process  (signal-name, signal-name, ..., signal-name)
type declarations
variable declarations
constant declarations
function definitions
procedure definitions
begin
sequential-statement
...
sequential-statement
end process;
```

Обратите внимание на то, что внутри процесса можно объявлять только «переменные», но не сигналы. *Переменная (variable)* в языке VHDL отслеживает состояние процесса только внутри него и вне процесса ее не видно. В зависимости от того, как используется переменная, ей в конце концов будет или не будет соответствовать определенный сигнал при физической реализации создаваемой схемы. Синтаксис определения переменной внутри процесса подобен синтаксису объявления сигнала в архитектуре, за исключением того, что используется ключевое слово *variable:*

```
variable variable-names : variable-type;
```

VHDL-процесс всегда либо *выполняется (running process),* либо *приостановлен (suspended process).* Перечнем сигналов в определении процесса, который называется *списком чувствительности (sensitivity list),* задаются условия, когда процесс выполняется. Первоначально процесс остановлен; когда изменяется значение любого из сигналов в его списке чувствительности, исполнение процесса

возобновляется, начиная с его первого последовательного оператора, и оно продолжается, пока не будет достигнут конец. Если какой-либо сигнал из списка чувствительности изменяет свое значение в результате исполнения процесса, то процесс выполняется снова. Это продолжается до тех пор, пока запуск процесса не перестанет приводить к изменению значения любого из этих сигналов. При моделировании все это происходит за нулевое время в модели.

Список чувствительности является необязательным; при моделировании исполнение процесса, у которого нет списка чувствительности, начинается в нулевой момент времени. В языке VHDL имеются последовательные операторы нескольких видов. Первый из них – это *последовательный сигнальный оператор присваивания (sequential signal-assignment statement);* у него тот же самый синтаксис, что и у параллельного аналога *(signal-name <= expression;),* но последовательный оператор располагается в теле процесса, а не в теле архитектуры. Аналогичный оператор для переменных – это *оператор присваивания значения переменной (variable-assignment statement),* синтаксис которого имеет вид *"variable-name : = expression;".* Заметьте, что в случае переменных используется другой оператор присваивания, а именно : =.

Ради учебных целей потоковая архитектура устройства для обнаружения простых чисел из листинга 16 воспроизведена в листинге 21 как процесс. Обратите внимание, что мы все еще продолжаем совершенствовать архитектуру того же самого объекта prime, который первоначально был объявлен в листинге 11. В этой новой архитектуре (prime6_arch) имеется только один параллельный оператор; этим оператором является процесс. Список чувствительности процесса содержит только массив N, представляющий собой первичные сигналы на входах устройства, реализующего желаемую комбинационную логическую функцию. Выходы вентилей И необходимо задать как переменные, а не как сигналы, поскольку внутри процесса определения сигналов не разрешены.

Листинг 21 – Потоковая VHDL-архитектура устройства обнаружения простых чисел, основанная на использовании процесса

```
architecture prime6_axch of prime is
begin
process(N)
variable N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC; begin
N3L_N0    := not N(3)                                 and N(0);
N3L_N2L_N1 := not N(3) and not N(2)  and     N(1)          ;
N2L_N1L_N0 :=              not N(2)  and     N(1)  and N(0);
N2_N1L_N0  :=                  N(2)  and not N(1)  and N(0);
F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
end process;
end prime6_arch;
```

Другие последовательные операторы помимо простого присваивания дают возможность творчески подойти к описанию поведения схемы. По-видимому, самый знакомый из них – это *оператор if (if statement),* синтаксис которого приведен в листинге 22. В первой и простейшей форме этого оператора проверяется булево выражение *boolean-expression*, если оно имеет значение true,

то исполняется последовательный оператор *sequential-statement.* Во второй форме добавляется предложение "else" с другим последовательным оператором *sequential-statement,* который исполняется, если булево выражение имеет значение false.

Листинг 22 – Синтаксис оператора if в языке VHDL

```
if boolean-expression then sequential-statement
end if;

if boolean-expression then sequential-statement
elsif boolean-expression then sequential-statement
elsif boolean-expression then sequential-statement
else sequential-statement
end if;
```

В листинге 23 представлен вариант архитектуры устройства для обнаружения простых чисел, в котором используется оператор if. Локальная переменная NI введена для того, чтобы отобразить преобразованное значение входного воздействия N в виде целого числа; это позволяет оперировать целыми числами при сравнениях в операторе if. Когда нужно выбирать среди нескольких альтернатив на основании значения только одного сигнала или выражения, обычно более читабельным и дающим лучший результат синтеза является оператор case (*casestatement*).

Листинг 23 – Архитектура устройства для обнаружения простых чисел, в которой использован оператор if

```
architecture prime7_arch of prime is
begin
process(N)
variable NI: INTEGER;
begin
NI := CONV_INTEGER(N);
if NI=I or NI=2 then F <= 'l';
elsif NI=3 or NI=5 or NI=7 or NI=11 or NI=13 then F <= 'l';
else F <=  '0';
end if;
end process;
end priffle7_arch;
```

Синтаксис оператора case представлен в листинге 24. В этом операторе вычисляется заданное выражение *expression,* по его значению выбирается одна из альтернатив *choices* и исполняются соответствующие последовательные операторы *sequential-statements.* Заметьте, что в каждом из наборов альтернатив *choices* можно записать один или большее число последовательных операторов. Сами альтернативы *choices* могут иметь форму одного значения или нескольких значений, разделенных вертикальной чертой (|). Альтернативы *choices* должны быть взаимно исключающими и содержать все возможные значения типа выражения *expression;* в последней альтернативе *choices* можно воспользоваться ключевым словом others для указания всех значений, которые еще не были упомянуты ранее.

Листинг 24 – Синтаксис оператора case в языке VHDL

```
case expression  is
when choices => sequential-statements
...
when choices => sequential-statements
end case;
```

Другой важный класс последовательных операторов образуют *операторы loop (loop statements).* Синтаксис простейшего из них указан в листинге 25. В этом примере возникает бесконечный цикл.

Листинг 25 – Синтаксис основного оператора loop в языке VHDL

```
loop
sequential-statement
...
sequential-statement
end loop;
```

Имеются еще два других полезных последовательных оператора, которые могут исполняться внутри цикла; это операторы exit (*exit statement*) и next (*next statement*). Оператор exit, когда он исполняется, передает управление оператору, непосредственно следующему за концом цикла. С другой стороны, оператор next вызывает пропуск всех остающихся в цикле операторов и переход к началу следующей итерации данного цикла. Устройство для обнаружения простых чисел вновь представлено в листинге 26, на этот раз – в виде архитектуры, в которой использован цикл for. Замечательно то, что в данном примере дается истинно поведенческое описание: здесь язык VHDL на самом деле используется для определения того, является входное воздействие N простым числом или нет. Мы увеличили также размерность массива N до 16 разрядов, чтобы подчеркнуть тот факт, что мы способны теперь создавать компактные модели схем, не перечисляя в явном виде сотни простых чисел.

Листинг 26 – Архитектура устройства для обнаружения простых чисел, в которой использован оператор for

```
library IEEE;
use IEEE.std_logic_1164.all;
entity pnme9 is
port ( N: in STD_LOGIV_VECTOR (15 downto 0);
       F: out STD_LOGIC);
end prime9;
architecture prime9_arch of prime9 is begin
process(N)
variable NI:   INTEGER;
variable prime:  boolean;
 begin
NI    := CONV_INTEGER(N);
prime := true;
if NI=1 or NI=2 then null;  — take care of boundary cases
 else for i in 2 to 253 loop
 if NI mod i=0 then
 prime := false;  exit;
 end if;
 end loop;
```

```
        end if;
 if prime then F <= '1'; else F <= '0'; end if;
end process;
end prime9_arch;
```

Оператор loop последнего вида – это цикл while (while loop). В такой конструкции булево выражение boolean-expression вычисляется перед началом каждой итерации, и цикл выполняется только до тех пор, пока значение этого выражения остается равным true.

ЛАБОРАТОРНАЯ РАБОТА №1
ИЗУЧЕНИЕ СРЕДСТВ УПРАВЛЕНИЯ ПРОЕКТИРОВАНИЕМ

Цель работы: усвоить навыки работы с системой WEBPACK ISE, научиться создавать проекты, проводить отладку, компиляцию и тестирование проекта.

1.1 Начало работы с системой WEBPACK ISE

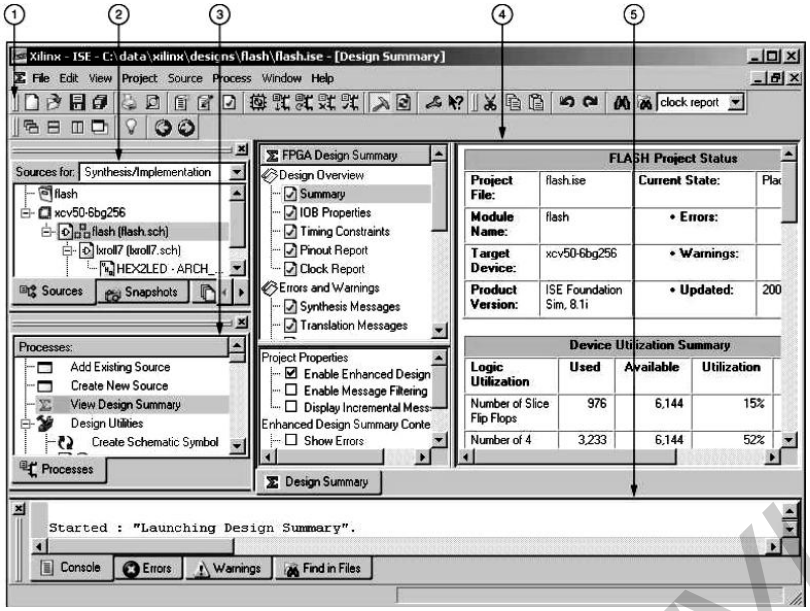
Для запуска САПР *WebPACK ISE* необходимо запустить «Навигатор проекта» *(Project Navigator)* – основную программу САПР *WebPACK ISE*. «Навигатор проекта» позволяет упорядочить файлы с исходным описанием проектируемого устройства, тестовыми модулями, модулями временных и топологических ограничений, а также предоставляет возможность простого доступа ко всем процессам, необходимым при проектировании цифрового устройства на базе ПЛИС с архитектурами *FPGA* и *CPLD*. На рисунке 1.1 показан внешний вид основного окна «Навигатора проекта» и его компоненты.

В процессе разработки цифрового устройства на базе ПЛИС в общем случае можно выделить следующие этапы.

1 Создание нового проекта, включающее выбор семейства и типа ПЛИС, а также средства синтеза.
2 Подготовка описания устройства в схемотехнической, текстовой или алгоритмической форме.
3 Синтез устройства.
4 Функциональное моделирование и тестирование.
5 Размещение и трассировка проекта в кристалле.
6 Временное моделирование.
7 Программирование ПЛИС (загрузка проекта в кристалл).

Каждому из этих этапов соответствует определенный набор процессов, к которым можно получить доступ из «Навигатора проектов».

Во время работы с САПР *WebPACK ISE* по всем необходимым вопросам рекомендуется обращаться к справочной системе самого *ISE* и сопутствующих продуктов. Для доступа к справочной системе достаточно нажать клавишу F1, что даст возможность получения контекстно-зависимой справки по выполняемой в настоящий момент задаче.



1 – инструментальная панель *(Toolbar);* 2 – окно описания проекта *(Source window);*
3 – окно процессов *(Process window);* 4 – рабочий стол *(Workspace);*
5 – окно отчетов *(Transcript window)*

Рисунок 1.1 – Основное окно «Навигатора проекта»

1.2 Создание нового проекта

Для создания нового проекта необходимо проделать ряд шагов.

1 Выбрать в меню пункт **File ► New Project...**, после чего будет запущен мастер нового проекта **New Project Wizard**.

2 Ввести или выбрать местоположение создаваемого проекта в поле **Project Location**.

3 Ввести имя проекта в поле **Project Name**. Обратите внимание на то, что к выбранному ранее местоположению проекта автоматически добавился каталог с введенным именем проекта. Этот каталог будет создан автоматически.

4 Убедиться, что в качестве типа основного файла описания устройства (**Top-Level Source Type**) выбран **HDL**.

5 Нажать кнопку **Next >** для перехода к странице выбора кристалла.

6 Заполнить свойства проектируемого устройства, как показано на рисунке 4:

– категория устройства *(Product Category):* **All;**
– семейство *(Family):* **Spartan-3AN;**
– устройство *(Device):* **XC3S700;**
– градация по быстродействию *(Speed):* -4;
– инструмент синтеза *(Synthesis Tool):* **XST (VHDL/Verilog)**;
– симулятор *(Simulator):* **ISE Simulator (VHDL/Verilog).**

7 Убедиться, что установлена опция **Enable Enhanced Design Summary**. В остальных полях необходимо оставить значения «по умолчанию».

8 Нажать кнопку **Next >** для перехода к странице создания заготовки основного файла с описанием устройства. После создания этого файла процесс создания проекта будет закончен.

1.3 Создание описания устройства

Проект может включать от одного до нескольких файлов с описанием проектируемого цифрового устройства. Описания можно создавать в виде электрических принципиальных схем, в виде *HDL-описания* на языке *VHDL* или *Verilog* или диаграмм состояний и переходов между ними. Кроме того, допускаются смешанные способы описания, представляющие собой сочетание перечисленных форм. В этом разделе остановимся на создании *HDL-описания* и использовании готовых шаблонов устройств.

Для создания *HDL-описания* устройства на языке *VHDL* необходимо выполнить следующие шаги.

1 В окне мастера нового проекта нажать кнопку **New Source.**

2 Выбрать *VHDL Module* в качестве типа исходного файла (рисунок 1.2).

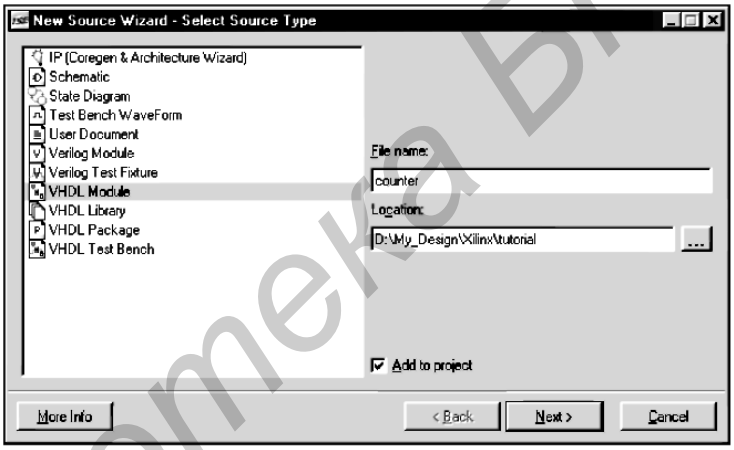


Рисунок 1.2 – Выбор типа файла

3 Ввести имя файла в поле *File name.*

4 Убедиться, что установлена опция *Add to project.*

5 Нажать кнопку **Next >**.

6 В открывшемся окне установить свойства портов ввода – вывода проектируемого устройства, как показано на рисунке 1.3.

7 Нажать кнопку **«Next >»**. **В** новом окне проверить свойства создаваемого файла с описанием устройства, после чего нажать кнопку **Finish.**

8 В следующих двух окнах нажать кнопку **Next >**. Появится окно с резюме о новом проекте. Если все свойства соответствуют требованиям, предъявляемым к проектируемому устройству, нажать кнопку **Finish.**

Таким образом, мы получили файл описания устройства с определенным разделом **entity** и незаполненным разделом **architecture.** Следующим шагом является создание архитектуры проектируемого устройства. При этом можно воспользоваться библиотекой шаблонов, которая включена в состав САПР *Web PACK ISE.*

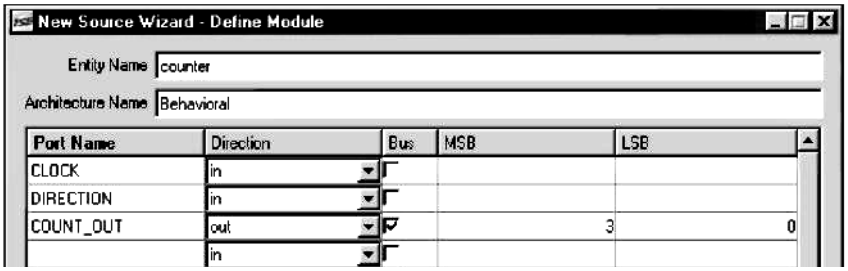


Рисунок 1.3 – Настройка портов ввода – вывода

Создадим при помощи библиотеки шаблонов простейший двоичный четырехразрядный счетчик. Для этого необходимо выполнить следующие шаги.

1 Поместить курсор после ключевого слова **begin** в разделе **architecture.**

2 Открыть библиотеку шаблонов *(Language Templates),* выбрав в меню пункт **Edit –► Language Templates....**

3 **В** открывшемся окне, используя символ «+», выбрать шаблон счетчика: **VHDL –► Synthesis Constructs –► Coding Examples –► –► Counters –► Binary –► Up/Down Counters –► –► Simple Counter.** В правой части окна появится текст шаблона на языке *VHDL.*

4 Для вставки шаблона в редактируемый файл выбрать пункт меню **Edit –► Use in file** или нажать кнопку на инструментальной панели.

Полученный код необходимо скорректировать. Для получения окончательного описания счетчика необходимо внести следующие правки.

1 Добавить описание промежуточного сигнала, предназначенного для переноса выходной информации от процесса, выполняющего функцию счетчика, к процессу формирующего требуемые значения на выходе устройства. Описание сигнала располагается в разделе architecture до ключевого слова begin:

signal count_int : std_logic_vector (3 downto 0) := "0000";

2 Изменить имена входных и выходных сигналов, описанных в шаблоне счетчика:

- заменить все <clock> на CLOCK;
- заменить все <count_direction> на DIRECTION;
- заменить все <count> на count_int.

3 Добавить после ключевого слова end process строку COUNT_OUT <= count_int;

4 Сохранить файл, выбрав в меню пункт **File ► Save** или нажать кнопку на инструментальной панели.

После окончания редактирования *HDL*-описания устройства необходимо проверить файл на наличие синтаксических ошибок. Для этого необходимо сделать ряд шагов.

1 Удостовериться, что в выпадающем меню **Sources for:** в окне описания проекта выбран режим *Synthesis/Implementation*.

2 Выбрать в этом же окне файл с описанием устройства **counter,** в окне процессов будут отображены процессы, доступные для этого устройства и режима.

3 Нажать символ «+» и раскрыть группу процессов *Synthesize-XST*.

4 Выполнить двойной щелчок мышью по пункту **Check Syntax.**

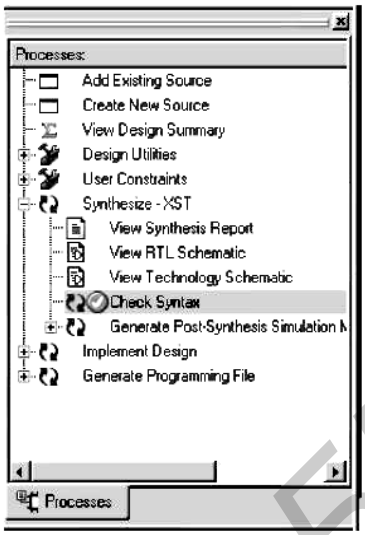


Рисунок 1.4 – Результат проверки синтаксиса

Сообщения обо всех найденных ошибках отображаются в разделе *Console* окна отчетов. Свидетельством отсутствия ошибок будет символ перед названием выполненного процесса. Если в результате работы процесса были обнаружены ошибки, то перед его именем появится символ (рисунок 1.4).

1.4 Тестовые модули и функциональное моделирование

Этап функционального моделирования *(Simulate Behavioral VHDL Model)* позволяет выполнить предварительную верификацию проекта. На этой стадии отсутствует информация о значениях задержек распространения сигналов, поэтому при функциональном моделировании можно обнаружить только логические и синтаксические ошибки в описании разрабатываемого устройства. Для функционального моделирования проекта применяется библиотека *UniSim Library,* элементы которой имеют единичные задержки.

1.4.1 Создание набора тестовых воздействий

При функциональном моделировании создается набор тестовых воздействий *(Test Bench),* по реакции на которые оценивается правильность работы проектируемого устройства. Для создания набора тестовых воздействий необходимо проделать следующие шаги.

1 Выбрать в окне описания проекта файл counter.

2 Для создания нового набора тестовых воздействий выбрать пункт меню **Project ► New Source.**

3 В открывшемся окне выбрать тип исходного файла *Test Bench WaveForm* и ввести имя для создаваемого набора counter_tbw в поле **File Name.**

4 Нажать кнопку **Next >**.

5 В открывшемся окне привязок *(Associate Source)* выбрать файл с описанием устройства, к которому будет привязан набор тестовых воздействий. Поскольку в нашем проекте всего один файл, то он уже выбран для создания связи.

6 Нажать кнопку **Next >**.

7 В окне резюме проверить соответствие файла с исходным описанием устройства и привязанного к нему набора тестовых воздействий, после чего нажать кнопку **Finish.**

8 В окне инициализации системы синхронизации *(Initialize Timing)* необходимо настроить временные параметры тактового сигнала.

Зададимся следующими требованиями к разрабатываемому счетчику:

– счетчик должен работать корректно при частоте входного сигнала 25 МГц;

– значение сигнала на входе выбора направления счета DIRECTION должно быть установлено не позднее чем за 10 не до установления высокого уровня на тактовом входе CLOCK;

– значение выходного сигнала на шине C0UNT_0UT должно быть установлено не позднее чем через 10 с после установления высокого уровня на тактовом входе CLOCK.

Для реализации этих требований необходимо заполнить поля настройки системы синхронизации следующим образом (рисунок 1.5):

– длительность высокого уровня тактового сигнала (Clock Time High) 20 нс;
– длительность низкого уровня тактового сигнала (Clock Time Low) 20 нс;
– время установления входного сигнала (Input Setup Time) 10 нс;
– задержка до установления выходного сигнала (Output Valid Delay) 10 нс;
– сдвиг (Offset) 0 нс;
– глобальный сигнал синхронизации (Global Signal): GSR (FPGA);
– длительность моделирования *(Initial Length of Test Bench)* **1500 нс.**

Значения остальных параметров оставить без изменений.

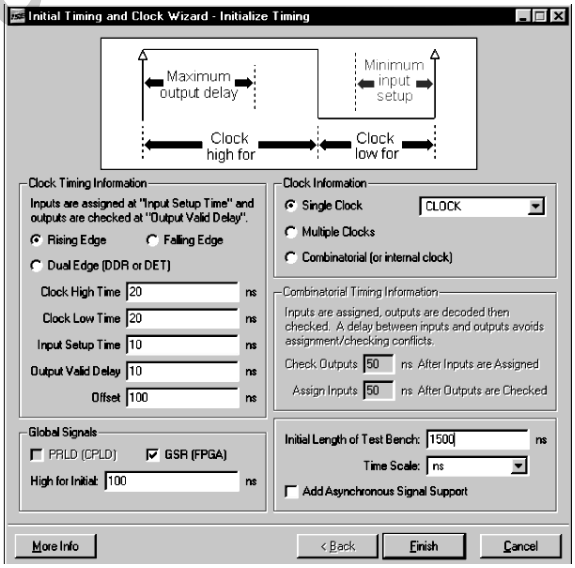


Рисунок 1.5 – Настройка системы синхронизации

9 Нажать кнопку **Finish** для окончания процесса настройки системы синхронизации.

10 На временной диаграмме (рисунок 1.6) в строке DIRECTION серым цветом отмечены зоны, равные по длительности заданному времени установления значения входного сигнала. Для моделирования обоих режимов работы счетчика (инкремент и декремент) установить точки переключения сигнала установки направления счета DIRECTION:

– щелкнуть мышью по голубой зоне в районе отметки 300 нс для установки высокого уровня сигнала DIRECTION и инкрементного режима работы счетчика;

– щелкнуть мышью по серой зоне в районе отметки 900 нс для установки низкого уровня сигнала DIRECTION и декрементного режима работы счетчика.

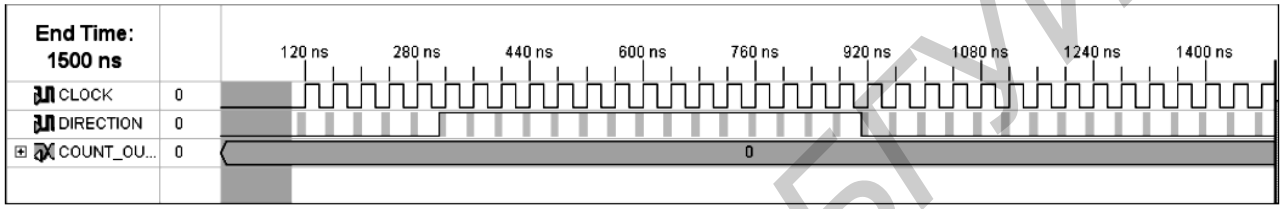


Рисунок 1.6 – Временная диаграмма набора тестовых воздействий

Временная диаграмма (*Waveform)* с заданными параметрами входных сигналов должна выглядеть, как показано на рисунке 1.5. Серая зона в начале диаграммы соответствует параметру «Сдвиг».

При установке глобального сигнала синхронизации к значению, заданному в поле *Offset*, автоматически добавляется 100 с, что и отражено на диаграмме.

11 Сохранить временную диаграмму.

12 В выпадающем меню **«Sources for:»** в окне описания проекта выбрать режим **«Behavioral Simulation»** (рисунок 1.7).

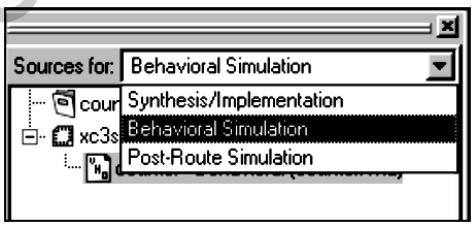


Рисунок 1.7 – Выбор режима функционального моделирования

1.4.2 Создание тестового модуля

Создание набора ожидаемых значений выходных сигналов завершает процесс создания набора тестовых воздействий. Эта операция преобразует набор тестовых воздействий в полноценный тестовый модуль. Целью создания такого модуля является получение возможности сравнения ожидаемых значений выходных сигналов со значениями, полученными в результате моделирования проектируемого устройства. По результатам такого моделирования делается заключение о правильности функционирования реализованного алгоритма.

Создавать тестовый модуль можно как вручную, так и автоматически.

Для создания тестового модуля в автоматическом режиме необходимо выполнить следующие шаги.

1 Убедиться, что выбран режим *Behavioral Simulation* в выпадающем меню **Sources for:** в окне описания проекта.

2 В окне описания проекта выбрать файл с тестовым модулем counter_tbw.

3 В окне процессов развернуть группу «Xilinx ISE Simulator» и выполнить двойной щелчок по процессу «Generate Expected Simulation Results». Запустится процесс моделирования работы проектируемого устройства.

4 В ответ на запрос диалогового окна «Expected Results» ответить **Yes**, будет выведена временная диаграмма с рассчитанными значениями выходных сигналов (рисунок 1.8).

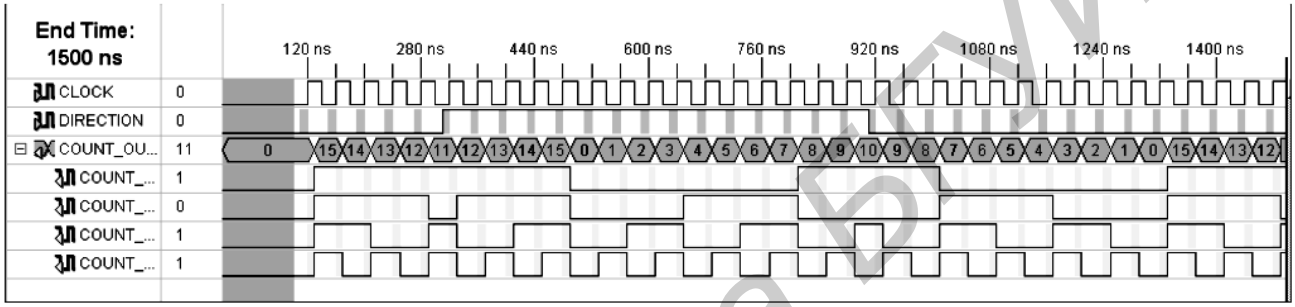


Рисунок 1.8 – Тестовый модуль, созданный автоматически

5 Для шины существует возможность просмотра временных диаграмм изменения сигнала на каждой из линий шины. Для этого необходимо щелкнуть по символу « + » перед именем шины (в нашем случае – COUNT_OUT).

При ручном режиме значения выходных сигналов редактируются точно так же, как и входных. Зоны установки значений выходных сигналов отмечены на временной диаграмме светло-серым цветом. После установки значений выходных сигналов необходимо выполнить процесс «Generate Expected Simulation Results», при этом в окне описания проекта должен быть выбран файл с тестовым модулем counter_tbw. В ответ на запрос диалогового окна «Expected Results» ответить **No;** будет выведена временная диаграмма с заданными и рассчитанными значениями выходных сигналов.

1.4.3 Функциональное моделирование

Получить информацию о работе устройства можно при помощи функционального моделирования. Этот процесс похож на процесс создания тестового модуля, но с той лишь разницей, что значения выходных сигналов не задаются, а только рассчитываются. Процесс функционального моделирования включает следующие шаги:

1 Убедиться, что выбран режим *Behavioral Simulation* в выпадающем меню **Sources for:** в окне описания проекта и выбран файл с тестовым модулем **counter_tbw.**

2 В окне процессов развернуть группу **Xilinx ISE Simulator** и выполнить двойной щелчок по процессу **Simulate Behavioral Model.**

3 Результаты моделирования выводятся в окне **Simulation** и доступны только для просмотра или печати (рисунок 1.9).

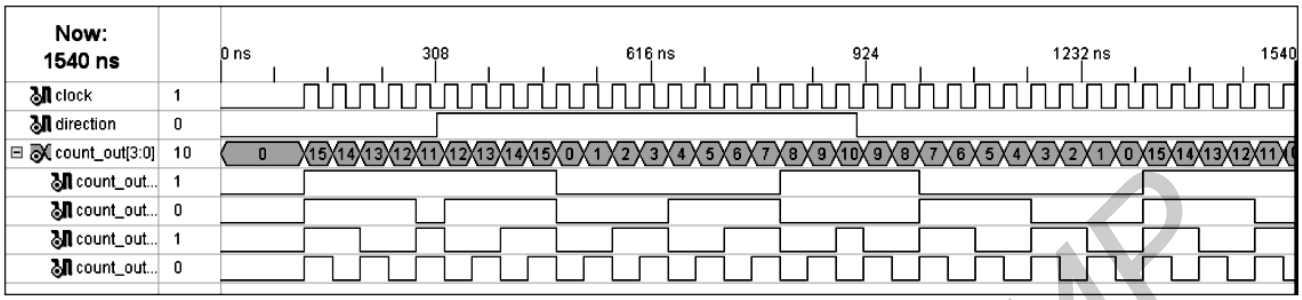


Рисунок 1.9 – Результаты функционального моделирования

1.5 Использование временных ограничений

В модулях временных и топологических ограничений содержится дополнительная информация для программы-трассировщика, размещающей спроектированное устройство на кристалле ПЛИС. Например, удобно задавать таким образом параметры глобального тактового сигнала, а также устанавливать соответствие между линиями портов проектируемого устройства и выводами ПЛИС. Кроме того, чтобы внести корректировки в параметры ограничений, например при изменении типа ПЛИС, достаточно скорректировать только модуль временных и топологических ограничений, а не редактировать различные файлы исходного описания.

Редактирование модуля временных и топологических ограничений производится при помощи специальной программы *Constraints Editor*. Для создания модуля необходимо продолжить следующие шаги.

1 В выпадающем меню **Sources for:** в окне описания проекта выбрать режим **Synthesis/Implementation.**

2 Выбрать в окне описания проекта файл counter.

3 Щелкнуть по символу «+», чтобы раскрыть группу процессов **User Constraints,** и дважды щелкнуть по процессу **Create Timing Constraints.** Автоматически запустятся процессы синтеза и трансляции устройства, после чего будет выдан запрос на создание и добавление в проект файла с временными и топологическими ограничениями (*User Constraints File)*.

4 В ответ на запрос ответить **Yes,** будет создан и добавлен в проект *UCF-файл* с именем **counter.ucf** и автоматически запустится редактор ограничений *(Xilinx Constraints Editor).*

5 Далее будем настраивать параметры тактового сигнала. Для этого необходимо выбрать вкладку *Global,* а на ней сигнал CLOCK и дважды щелкнуть по полю *Period.*

6 В поле *Time* установить значение 40 нс**.** Остальные поля оставить без изменений.

7 Нажать кнопку **ОК**.

8 На вкладке *Global* выбрать сигнал CLOCK и дважды щелкнуть по полю *Pad to Setup* для запуска мастера настройки времени установки значений сигналов на выходах проектируемого устройства.

9 В поле *Offset* ввести значение 10 нс.

10 Нажать кнопку **ОК**.

11 На вкладке *Global* выбрать сигнал CLOCK и дважды щелкнуть по полю *Clock to Pad* для запуска мастера настройки времени реакции на синхросигнал.

12 В поле *Offset* ввести значение 10 нс.

13 Нажать кнопку **ОК**. Подготовленный файл настройки должен выглядеть, как показано на рисунке 1.9.

Далее необходимо сохранить созданный файл ограничений и, закрыв редактор, вернуться в «Навигатор проектов».

1.6 Реализация устройства на базе ПЛИС

Этап реализации *(Implementation)* проектов, выполняемых на базе ПЛИС с архитектурой *FPGA,* включает в себя три фазы: трансляции (*Translate),* отображения логического описания проекта на физические ресурсы кристалла *(Map),* размещения и трассировки *(Place and Route).* В результате выполнения этого этапа создается двоичный файл, который описывает использование физических ресурсов кристалла для реализации элементов (функций) проектируемого устройства и выполнение необходимых соединений между ними. Этот файл затем используется в качестве исходного для генерации конфигурационной последовательности.

1.6.1 Размещение устройства в кристалле

Процесс размещения устройства на кристалле включает следующие шаги.

1 Выбрать в окне описания проекта файл counter.

2 Открыть окно резюме проекта, дважды щелкнув по процессу *View Design Summary* в окне процессов.

3 Запустить процесс реализации проекта, дважды щелкнув по процессу *Implement Design* в окне процессов.

4 Если процесс реализации проекта прошел без ошибок и предупреждений, то это будет отмечено зеленым символом перед именем процесса. Если же произошел сбой на одном из этапов работы, то этот этап можно идентифицировать по красному (признак ошибки) или желтому (признак предупреждения) символу перед именем процесса *Implement Design,* а также перед именем сбойного подпроцесса.

5 В нижней части окна резюме проекта *(Design Summary)* найти раздел *Performance Summary,* включающий характеристики производительности спроектированного устройства.

6 Щелкнуть по ссылке, ведущей в подраздел *All Constraints Met,* откроется отчет о временных ограничениях, реализованных в конечном устройстве (рисунок 1.10).

7 Сравнить полученные временные ограничения с теми, которые мы задавали в *UCF-файле.*

Met	Constraint	Requested	Actual	Logic Levels	Absolute Slack	Number of errors
Yes	OFFSET = OUT 10 ns AFTER COMP "CLOCK"	10.000ns	9.096ns	1	0.904ns	0
Yes	OFFSET = IN 10 ns BEFORE COMP "CLOCK"	10.000ns	1.141ns	3	8.859ns	0
Yes	TS_CLOCK = PERIOD TIMEGRP "CLOCK" 40 ns HIGH 50%	40.000ns	3.477ns	0	36.523ns	0

Рисунок 1.10 – Отчет о временных ограничениях проекта

1.6.2 Назначение выводов

Если необходимо поставить в однозначное соответствие линии портов проектируемого устройства выводам микросхемы ПЛИС, используются соответствующие настройки в *UCF-файле.*

Для описания соответствующих топологических ограничений необходимо выполнить следующие шаги.

1 Выбрать в окне описания проекта файл counter.

2 Дважды щелкнуть по имени процесса *Assign Package Pins*, расположенного в группе процессов *User Constraints*. Откроется редактор *PACE (Pinout and Area Constraints Editor).*

3 Выбрать вкладку *Package View.*

4 В окне списка объектов проекта *(Design Object List)* ввести в поле *Loc* местоположение линий портов в ПЛИС, согласно заданию преподавателя:

5 Выбрать пункт меню **File ► Save.** На вопрос об используемом **разделителе** при описании шин необходимо ответить **XST Default:** <> и нажать **ОК**.

6 Закрыть редактор *PACE.*

После установления соответствия между линиями портов проекта и выводами микросхемы ПЛИС группа процессов *Implement Design* будет помечена оранжевой отметкой, что свидетельствует об изменении условий размещения проекта. В нашем случае это касается изменений в *UCF-файле.*

1.7 Переразмещение проекта и проверка соответствия выводов

После изменений в назначении выводов микросхемы ПЛИС, рассмотренных в подразделе 1.6, необходимо выполнить переразмещение проекта счетчика на кристалле ПЛИС. Для этого необходимо удостовериться, что все выводы микросхемы правильно поставлены в соответствие портам проектируемого счетчика. Для этого необходимо:

1 Открыть резюме проекта *(Design Summary)*, дважды щелкнув по пункту **View Design Summary** в окне процессов.

2 Выбрать отчет об использовании выводов микросхемы *(Pinout Report)* и щелкнуть по заголовку колонки с именами сигналов *(Signal Name)*, чтобы выводы микросхемы были отсортированы по именам портов проекта.

3 Выполнить переразмещение проекта, дважды щелкнув по процессу *Implement Design.*

4 Обновить отчет об использовании выводов, щелкнув по пункту *View Design Summary* в окне процессов.

5 Удостовериться, что информация, приведенная в отчете, совпадает с требованиями к назначению выводов.

1.8 Проверка проекта способом временного моделирования

Временное моделирование после размещения проектируемого устройства на кристалле ПЛИС в отличие от функционального моделирования учитывает реальные задержки, возникающие при распространении сигналов в ПЛИС. Оно является завершающей стадией проектирования цифрового устройства. На этом этапе определяются время реакции системы на входные воздействия и другие временные параметры спроектированного цифрового устройства. Для выполнения временного моделирования необходимо выполнить следующие шаги.

1 В выпадающем меню **Sources for:** в окне описания проекта выбрать режим *Post-Route Simulation.*

2 В окне описания проекта выбрать файл с тестовым модулем counter_tbw.

3 Запустить процесс временного моделирования, дважды щелкнув по процессу *Simulate Post-Place Route Model* из группы процессов *Xilinx ISE Simulator.*

4 Убедиться, что при переключении сигнала DIRECTION изменяется направление счета (рисунок 1.12).

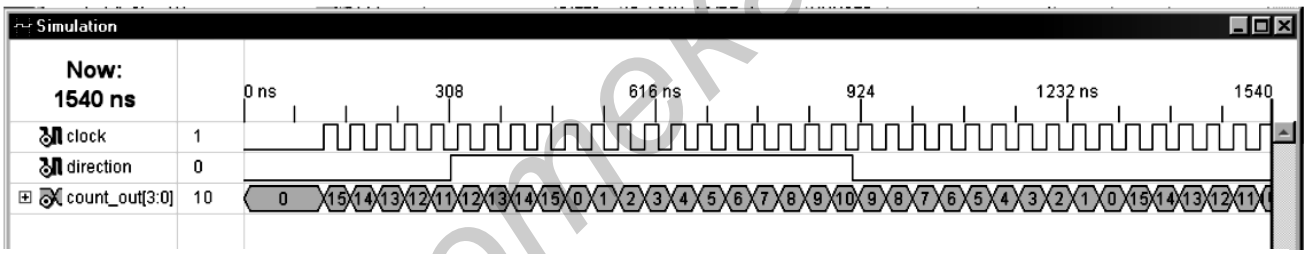


Рисунок 1.12 – Результаты временного моделирования

5 Убедиться, что вкладка *Errors* в окне отчетов не содержит сообщений об ошибках.

6 Увеличив масштаб нажатием кнопки на инструментальной панели, можно отследить время реакции системы от переднего фронта тактового сигнала **CLOCK** до изменения значения на выходе COUNT_OUT.

7 Закрыть окно временного моделирования.

1.9 Загрузка конфигурации в Spartan-3A Demo Board

После окончательной проверки работоспособности спроектированного устройства с использованием симулятора *ISE* можно протестировать цифровое устройство при помощи демонстрационной платы, например *Spartan™-3 Demo Board.* Для загрузки конфигурации в ПЛИС необходимо проделать следующие шаги.

1 Подключить блок питания с выходным напряжением 5 В постоянного тока к демонстрационной плате (J4).

2 Подключить загрузочный кабель к LPT-порту компьютера и разъему *JTAG* демонстрационной платы (J7).

3 В выпадающем меню **Sources for:** в окне описания проекта выбрать режим *Synthesis/Implementation.*

4 Выбрать в окне описания проекта файл counter.

5 В окне процессов щелкнуть по символу «+», чтобы раскрыть группу процессов *Generate Programming File.*

6 Дважды щелкнуть по процессу *Configure Device (iMPACT).* Будет запущена программа *iMPACT* и откроется окно выбора режима конфигурирования.

7 В открывшемся окне выбрать резким конфигурирования с использованием периферийного сканирования **Configure devices using Boundarj Scan** *(JTAG).*

8 Убедиться, что в выпадающем списке выбран пункт **Automatically connect to a cable and identify Boundary-Scan chain.**

9 Нажать кнопку **Finish.**

10 Если появится сообщение о том, что найдено два устройства, нажать **ОК** для продолжения работы. Будет сформирован канал периферийного сканирования в соответствии со стандартом **JTAG** и произойдет автоматическое определение устройства, после чего появится основное окно программы **iMPACT.**

11 В открывшемся диалоговом окне выбрать файл битовой последовательности counter.bit для загрузки в устройство xc3s200, после чего нажать кнопку **«Open»**.

12 Если появится предупреждение *(Warning)*, то необходимо нажать **ОК**.

13 Снова откроется окно выбора файла битовой последовательности. Поскольку в цепочке *JTAG* в нашем случае только один ПЛИС, необходимо нажать **Bypass,** чтобы пропустить загрузку в другие устройства.

14 Правой кнопкой мыши щелкнуть по устройству xc3s200 и выбрать в выпадающем меню пункт **Program...** Откроется диалоговое окно определения настроек программирования устройства *(Programming Properties).*

15 Нажать **ОК** для начала программирования. Об успешном окончании программирования будет свидетельствовать появившееся в главном окне *IMPACT* сообщение **Program Succeeded.**

16 Если появится предупреждение (*Warning*), то нажать **ОК**. О нормальной работе счетчика свидетельствуют горящие светодиоды LDO, LD1, LD2 и LD3.

ЛАБОРАТОРНАЯ РАБОТА №2
РАЗРАБОТКА ДЕШИФРАТОРА

Цель работы: овладеть знаниями и практическими навыками по проектированию дешифраторов в среде Xilinx ISE Design Suite 10.1, а также по программированию и отладке логических схем на языке VHDL.

2.1 Теоретические сведения

Дешифратор (decoder) – это логическая схема с несколькими входами и несколькими выходами, которая преобразует кодированные входные сигналы в кодированные выходные сигналы, причем входные и выходные коды различны.

Входной код обычно имеет меньшее число разрядов, чем выходной код, и между входными и выходными кодовыми словами имеется взаимно однозначное соответствие. При взаимно однозначном соответствии (one-to-one mapping) каждое входное кодовое слово порождает отличное от других выходное кодовое слово. Для того чтобы дешифратор нормально выполнял функцию отображения, необходимо подать сигналы на входы разрешения, если таковые имеются. Иначе дешифратор отображает все входные кодовые слова кода в единственное «запрещенное» выходное кодовое слово.

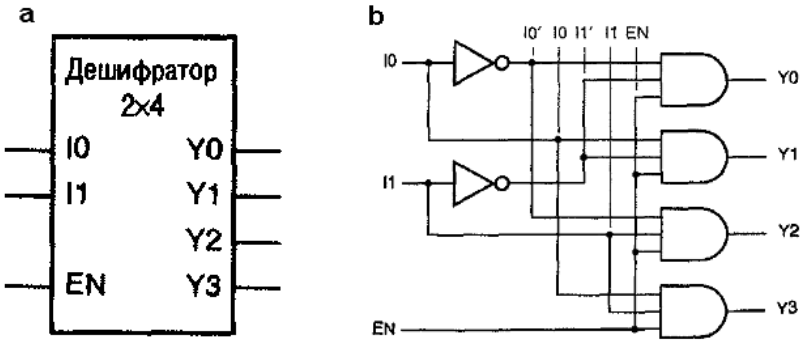
В большинстве случаев роль входного кода играет n-разрядный двоичный код, где n-разрядное двоичное слово представляет одну из 2^n различных кодированных величин. Обычно это целые числа от 0 до 2n–1. Иногда для представления меньшего, чем 2^n, числа величин применяют усеченный n-разрядный двоичный код. Например, в двоично-десятичном коде 4-разрядные комбинации от 0000 до 1001 представляют десятичные цифры от 0 до 9, а комбинации от 1010 до 1111 не используются.

В большинстве случаев роль выходного кода играет m-разрядный код «1 из m», у которого в любой момент времени отличен от нуля один бит. Таким образом, в коде «1 из 4» с высоким активным уровнем сигнала на выходах кодовые слова имеют вид 0001, 0010, 0100 и 1000. При низком активном уровне сигнала на выходах кодовые слова имеют вид 1110, 1101, 1011 и 0111.

Самым распространенным является дешифратор n x 2^n или полный дешифратор (binary decoder). На вход такого дешифратора поступает n-разрядное двоичное кодовое слово, а на выходе возникает слово кода «1 из 2^n». Полный дешифратор применяется в том случае, когда необходимо активизировать точно один из 2^n выходов, определяемый n-разрядным двоичным числом на входе.

На рисунке 2.1, а, например, указаны входы и выходы, а также приведена таблица истинности дешифратора 2x4. Входное кодовое слово 11,10 представляет собой целое число из интервала от 0 до 3. Выходные кодовые слова Y3, Y2, Y1, Y0 формируются по следующему правилу: Yi равно 1 только в том случае, когда входное кодовое слово является двоичным представлением числа и *входной сигнал разрешения (enable input)* EN равен 1. Если EN = 0, то на всех выходах устанавливается логический 0. На рисунке 2.1, б показана

схема дешифратора 2x4 на уровне вентилей. Каждый вентиль И *декодирует (decode)* одну комбинацию входного кодового слова I1, I0.



а – дешифратор 2x4 входы и выходы; б – принципиальная схема

Рисунок 2.1 – Дешифратор

В таблице истинности полного дешифратора среди входных комбинаций фигурирует символ «безразличного» значения (таблица 2.1). Если одно или большее число входных величин не влияют на значения выходных сигналов при какой-то определенной комбинации сигналов на остальных входах, то такие входные сигналы в данной комбинации отмечаются символом «x».

Таблица 2.1 – Таблица истинности для полного дешифратора 2x4

Входы		***Выходы***			
EN	I0	Y3	Y2	Y1	Y0
0	x	0	0	0	0
1	0	0	0	0	I
1 0	1	0	0	1	0
1 1	0	0	1	0	0
1 1	1	1	0	0	0

Общие рекомендации по графическому изображению крупных логических элементов:

— в условном обозначении этих элементов входы рисуются слева, а выходы – справа. К верхней и нижней стороне условного изображения сигналы обычно не подводятся. Однако иногда сверху и снизу в явном виде изображают выводы для подключения напряжения питания и земли;

— у больших логических элементов имена сигналов почти всегда выбираются по названиям функций, выполняемых *внутри* этих элементов;

— выводам с высоким активным уровнем сигнала дают то же самое имя, какое имеет внутренний сигнал, в то время как выводы с низким активным уровнем сигнала получают имя внутреннего сигнала с добавлением суффикса «L».

2.2 Описание дешифраторов на языке VHDL

Существует несколько подходов к проектированию дешифраторов на языке VHDL. В простейшем случае следовало бы записать структурный эквивалент принципиальной схемы дешифратора, как это сделано в листинге 2.1 для полного дешифратора 2x4, приведенного на рисунке 2.1. Предполагается, что компоненты and3 и inv уже существуют в микросхеме, для которой пишется программа. Однако такое механическое преобразование существующих устройств в эквивалент в виде списка соединений не раскрывает возможностей языка VHDL, очень усложняет программу и увеличивает время на ее разработку.

Листинг 2.1 – Структурная программа дешифратора, изображенного на рисунке 2.1, на языке VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V2to4dec is
     port (I0, II, EN: in STD_LOGIC;
           Y0, Yl, Y2, Y3: out STD_LOGIC );
end V2to4dec;

architecture V2to4dec_s of V2to4dec is
signal N0TI0, N0TI1; STD_LOGIC;
component inv port (I: in STD_LOGIC; O: out STD_LOGIC );
end component;
component and3 port (I0, I1, I2: in STD_LOGIC; O: out STD_LOGIC );
end component;
begin
Ul: inv port map (I0,NOTI0);
U2: inv port map (Il,NOTIl);
U3: and3 port map (N0TI0,NOTI1, EN, Y0);
U4: and3 port map (I0,   NOTIl, EN, Yl);
U5: and3 port map (NOTI0,I1,    EN, Y2);
U6: and3 port map (I0,   I1,    EN, Y3);
end V2to4dec_s;
```

Вместо этого необходимо написать программу, в которой язык VHDL был бы использован так, чтобы сделать проектирование дешифратора более понятным и удобным. В листинге 2.2 продемонстрирован один из подходов к написанию программы полного дешифратора 3x8 на языке VHDL в стиле потокового проектирования.

Адресные входы A (2 downto 0) и декодированные выходные сигналы Y (0 to 7) представлены в виде векторов. В операторе select перечислены восемь случаев декодирования, в каждом из которых 8-разрядному внутреннему сигналу Y_s присваивается соответствующая комбинация входных сигналов. Эта комбинация присваивается фактическому выходному сигналу схемы Y только в том случае, когда сигналы на всех входах разрешения (Gl, G2, G3) активны.

Листинг 2.2 – Определение дешифратора 3x8 с высокими активными уровнями сигналов в потоковом стиле

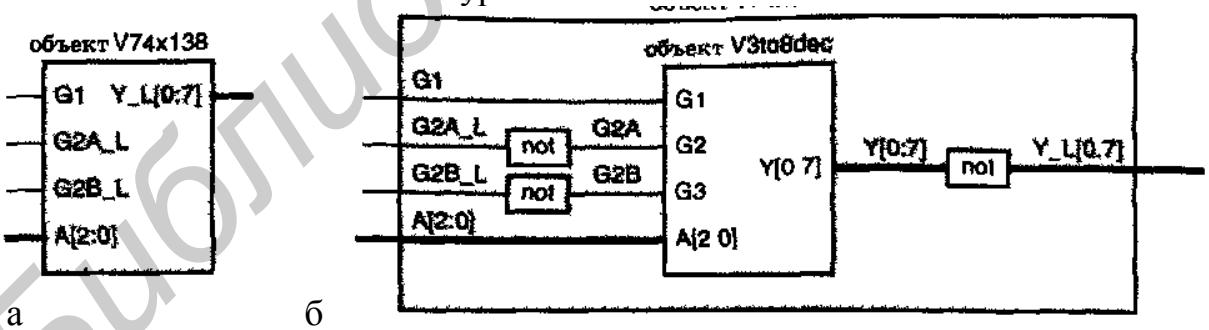
```
library IEEE;
use IEEE.std_logic_1164.all;

entity V3to8dec is
     port (Gl, G2, G3: in STD_L0GIC;
           A:          in STD_L0GIC_VECTOR (2 downto 0);
           Y:          out STD_LOGIC_VECTOR (0 to 7) );
end V3to8dec;

architecture V3to8dec_a of V3to8dec is
signal Y_s: STD_LOGIC_VECTOR (0 to 7);
begin
with A select Y_s <= "10000000" when "000",
"01000000" when "001",
"00100000" when "010",
"00010000" when "011",
"00001000" when "100",
"00000100" when "101",
"00000010" when "110",
"00000001" when "111",
"00000000" when others;
Y <= Y_s when (Gl and G2 and G3)='1' else "00000000";
end V3to8dec_a;
```

При необходимости на основе дешифратора 3x8 можно реализовать более сложные функции, например дешифратор V74x138, у которого выход Y_L и два разрешающих входа (G2A_L, G2B_L) имеют низкие активные уровни. Соотношение между объектами показано на рисунке 2.2.

Как показано в листинге 2.3, архитектуру V74x138 можно организовать иерархически, используя компонент V3to8dec, в котором фигурируют сигналы только с высокими активными уровнями.



а б

а – верхний уровень; б – внутренняя организация с архитектурой V74x138_c

Рисунок 2.2 – VHDL-объект V74x138

Листинг 2.3 – Иерархическое определение дешифратора типа 74x138 с преобразованием активных уровней

```
library IEEE;
use IEEE.std_logic_1164.all;
entity V74xl38 is
```

```
      port (Gl, G2A_L, G2B_L: in STD_L0GIC;   -- входы разрешения
            A: in STD.LOGIC_VECTOR (2 downto 0); -- входы управления
            Y_L: out STD_LOGIC_VECTOR (0 to 7) ); — выход дешифратора
end V74xi38;

architecture V74xl38_c of V74xi38 is
 signal G2A, G2B: STD_L0GIC;     -- промежуточные сигналы управления
 signal Y: STD_L0GIC_VECTOR (0 to 7)
 component V3to8dec
      port (Gl, G2, G3: in STD_LOGIC;
            A: in STD_LOGIC_VECTOR (2 downto 0);
            Y: out STD_LOGIC_VECTOR (0 to 7) );
 end component;
begin
 G2A <= not G2A_L;  --инверсия входного сигнала
 G2B <= not G2B_L;  -- для согласования с готовым компонентом
 Y_L <= not Y;      --инверсия выходного сигнала
 Ul: V3to8dec port map (Gl, G2A, G2B, A, Y); -- вызов компонента
end V74xl38_c;
```

2.3 Приборы и оборудование, применяемые в работе

1 Комплект разработки для ПЛИС Spartan 3AN.

2 Персональный компьютер с USB портом.

3 Программное обеспечение Xilinx ISE Design Suite 10.1.

2.4 Порядок выполнения работы

1 Запустить среду разработки Xilinx ISE Design Suite 10.1.

2 Разработать программу дешифратора в соответствии с выданным вариантом задания.

3 Скомпилировать программу, провести исправление ошибок.

4 Выполнить тестирование проекта.

5 Подготовить комплект отладки и ПК. Подключить плату к компьютеру по интерфейсу USB. Подать питание на плату с помощью 9-вольтного блока питания. Установить связь МК с компьютером.

6 Записать программу во внутреннюю память ПЛИС.

7 Проверить работу микросхемы.

8 Запустить работу (выключить питание платы, отсоединить плату, выключить компьютер).

Варианты заданий по выполнению лабораторной работы представлены в таблице 2.2.

Таблица 2.2 – Варианты заданий

Номер варианта	Тип дешифратора
1	Дешифратор 4x16
2	Дешифратор 5x16
3	Дешифратор для восьмисегментных индикаторов
4	Дешифратор для символьной индикации 5x8 пикселей

2.5 Содержание отчета

1 Цель работы.
2 Листинг программы генерации сигнала.
3 Окно моделирования работы программы.
4 Выводы.

2.6 Контрольные вопросы

1 Каково назначение и функции дешифратора?
2 Что такое «полный дешифратор»?
3 Зачем в таблице истинности используется символ «x»?
4 Назовите рекомендации по графическому изображению логических элементов.
5 Объясните структуру объявления дешифратора.
6 Поясните особенности архитектуры разработанного дешифратора.

ЛАБОРАТОРНАЯ РАБОТА №3
РАЗРАБОТКА МУЛЬТИПЛЕКСОРА

Цель работы: овладеть знаниями и практическими навыками по проектированию мультиплексоров, а также по программированию и отладке описания логических схем на языке VHDL.

3.1 Теоретические сведения

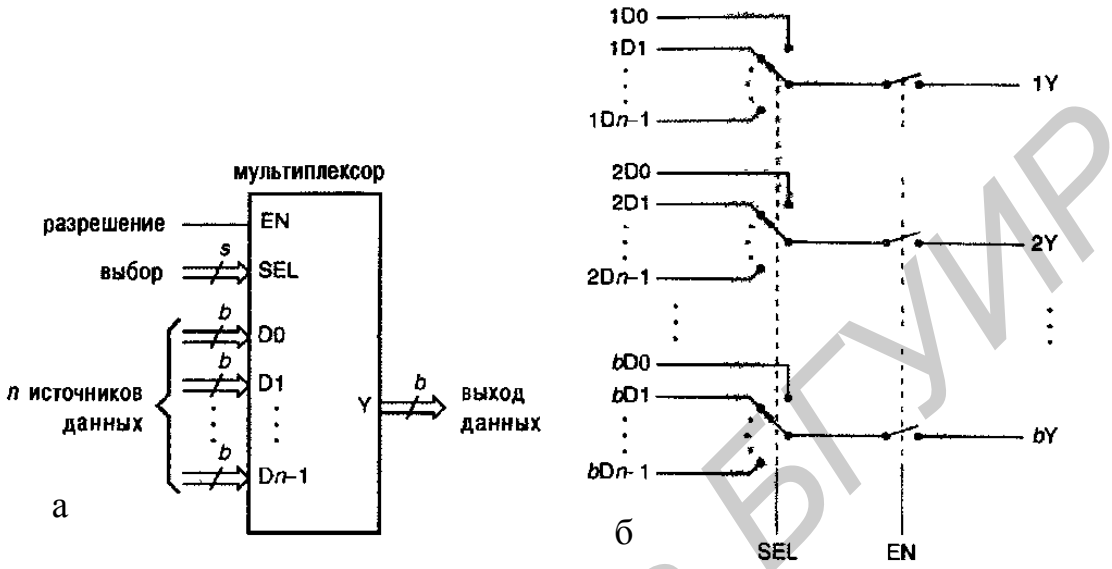
Мультиплексором (multiplexer) называется цифровой переключатель, который осуществляет передачу на выход данных, поступающих от одного из *n* источников. На рисунке 3.1, а изображены входы и выходы *n*-входового 6-разрядного мультиплексора. Имеются *n* источников *b*-разрядных данных и один *b*-разрядный выход. У типичных, выпускаемых серийно мультиплексоров *n* = 1, 2, 4, 8 или 16, а *b* = 1, 2 или 4. Имеются *s* адресных входов, с помощью которых выбирается один из *n* источников, поэтому $s = [\log_2 n]$. По сигналу на входе разрешения EN мультиплексор коммутирует цифровой сигнал со входа (адрес которого задан) на выход ; когда EN = 0, сигналы на всех выходах равны 0. По-английски, для краткости, мультиплексор часто называют *mux.*

На рисунке 3.1, б приведена схема переключения, являющаяся грубым эквивалентом мультиплексора. Но в отличие от механического переключателя мультиплексор является однонаправленным устройством: информационные потоки направлены только от входов (расположенных слева) к выходам (расположенным справа).

3.1.1 Мультиплексоры, демультиплексоры и шины данных

Чтобы выбрать один из *n* источников данных для передачи их по шине, можно воспользоваться мультиплексором. Чтобы направить данные к одному из n адресатов на приемном конце шины, можно применить демультиплексор (demultiplexer). Такая система с 1-разрядной шиной изображена в виде

переключателей на рисунке 3.2, а. В блок-схемах мультиплексоры и демультиплексоры часто изображают в виде трапеций (рисунке 3.2, б), чтобы наглядно показать, как сигналы одного выбранного из многих, источника данных поступают на шину и направляются к тому или иному адресату, выбранному из многих адресатов.



а – входы и выходы; б – функциональный эквивалент

Рисунке 3.1 – Структура мультиплексора

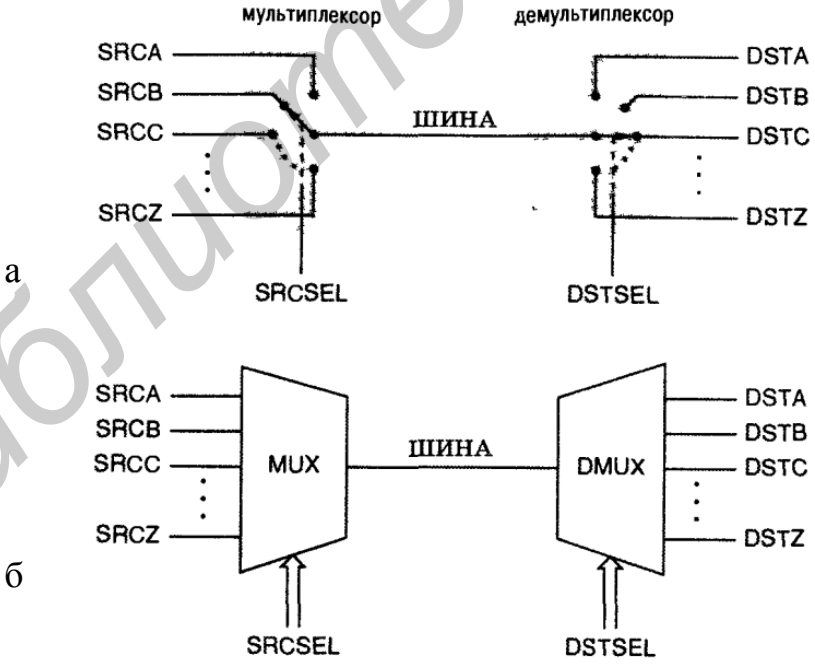


Рисунок 3.2 – Система с 1-разрядной шиной

Функция, реализуемая демультиплексором, прямо противоположна функции, выполняемой мультиплексором. Например, 1-разрядный демультиплексор с *n* выходами имеет один вход данных и *s* входов выбора

одного из $n = 2^s$ выходов данных. При нормальной работе сигналы на всех выходах, кроме выбранного, равны 0; данные на выбранном выходе совпадают с данными на входе. Это определение можно обобщить на 6-разрядный демультиплексор с n выходами; у такого демультиплексора b входов данных, и в нем с помощью s сигналов на входах адреса выбирается один из $n = 2^s$ наборов с b выходами данных в каждом.

3.2 Описание мультиплексоров на языке VHDL

Описывать мультиплексоры на языке VHDL достаточно просто. В архитектуре, написанной в потоковом стиле, оператор SELECT обеспечивает требуемые функциональные возможности, что можно видеть в листинге 3.1, где дано описание 4-входового 8-разрядного мультиплексора на языке VHDL.

В поведенческой архитектуре выбор осуществляется оператором CASE. В листинге 3.2, например, приведена архитектура для того же самого модуля mux4in8b, основанная на использовании процесса.

Листинг 3.1 – Потоковая VHDL-программа для 4-входового 8-разрядного мультиплексора

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4in8b
    is
       port (
       S: in STD_LOGIC_VECTOR (1 downto 0);
       A, B, C, D: in STD_LOGIC_VECTOR (1 to 8);
       Y: out STD_L0GIC_VECT0R (1 to 8) );
end mux4in8b;

architecture mux4irt8b of mux4in8b is
begin
    with S select Y <=
      A when "00",
      B when "01",
      C when "10",
      D when "11",
       (others => 'U') when others;
end mux4in8b;
```

Листинг 3.2 – Поведенческая архитектура для 4-входового 8-разрядного мультиплексора

```
architecture mux4in8p of mux4in8b is
begin
process(S, A, B, C, D)
  begin
    case S is
      when "00" => Y <= A;
      when "01" => Y <= B;
```

```
      when "10" => Y <= C;
      when "11" => Y <= D;
      when others => Y <= (others => 'U');
    end case;
  end process;
end mux4in8p;
```

В VHDL-программе мультиплексора очень просто реализовать любые требуемые правила выбора входа. Например, в листинге 3.3 приведена написанная в поведенческом стиле программа для специализированного 4-входового 18-разрядного мультиплексора. Если в каждом из этих примеров сигналы на входах выбора недействительны (например содержат значения 'U' или 'X'), то для того, чтобы облегчить обнаружение ошибок в процессе моделирования, сигналам на выходной шине присваиваются значения 'U'.

Листинг 3.3 – Поведенческая VHDL-программа для специализированного 4-входового 18-разрядного мультиплексора

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4in3b is
   port (
      S: in STD_L0GIC_VECTOR (2 downto 0);
      A, B, C, D: in STD_LOGIC_VECTOR (1 to 18);
      Y: out STD_L0GIC_VECTOR (1 to 18)
   );
end mux4m3b;

architecture mux4m3p of mux4in3b is
begin
process (S, A, B, C, D)
   variable Y: INTEGER;
   begin
    case S is
      when "000" | "010" | "100" | "110" => Y <= A;
      when "001" | "111" => Y <= B;
      when "011" => Y <= C;
      when "101" => Y <= D;
      when others => Y <= (others => 'U');
    end case;
  end process;
end mux4in3p;
```

3.3 Приборы и оборудование, применяемые в работе

1 Комплект разработки для ПЛИС Spartan 3AN.
2 Персональный компьютер с USB-портом.
3 Программное обеспечение Xilinx ISE Design Suite 10.1.

3.4 Порядок выполнения работы

1 Запустить среду разработки Xilinx ISE Design Suite 10.1.

2 Разработать программу мультиплексора в соответствии с выданным вариантом задания.

3 Скомпилировать программу, провести исправление ошибок.

4 Выполнить тестирование проекта.

5 Подготовить комплект отладки и ПК. Подключить плату к компьютеру по интерфейсу USB. Подать питание на плату с помощью 9В блока питания. Установить связь с компьютером.

6 Записать программу во внутреннюю память ПЛИС.

7 Проверить работу микросхемы.

8 Запустить работу (выключить питание платы, отсоединить плату, выключить компьютер).

Варианты заданий по выполнению лабораторной работы представлены в таблице 3.1.

Таблица 3.1 – Варианты заданий

Номер варианта	Тип мультиплексора
1	Мультиплексор
2	Мультиплексор + демультиплексор
3	Мультиплексор + демультиплексор+ дешифратор
4	Мультиплексор + демультиплексор + дешифратор

3.5 Содержание отчета

1 Цель работы.

2 Листинг программы мультиплексора.

3 Окно моделирования работы программы.

4 Выводы.

3.6 Контрольные вопросы

1 Каково назначение и функции мультиплексоров?

2 Как определяется разрядность шины адреса мультиплексора

3 Что такое демультиплексор?

4 Как используют мультиплексоры и демультиплексоры для передачи данных ?

5 Объясните структуру объявления мультиплексора.

6 Поясните особенности архитектуры разработанного мультиплексора.

ЛАБОРАТОРНАЯ РАБОТА №4
РАЗРАБОТКА АРИФМЕТИКО-ЛОГИЧЕСКОГО УСТРОЙСТВА

Цель работы: овладеть знаниями и практическими навыками по проектированию арифметико-логических устройств (АЛУ). Лабораторная работа также служит для овладения навыками программирования и отладки описания логических схем на языке VHDL.

4.1 Теоретические сведенья

Арифметико-логическое устройство (АЛУ; arithmetic and logic unit, ALU) является комбинационной схемой, способной выполнять целый ряд различных арифметических и логических операций с парой n-разрядных операндов. Выполняемая операция определяется комбинацией сигналов на входах выбора функции.

Классификация АЛУ:

По способу действия над операндами АЛУ делятся на последовательные и параллельные. В последовательных АЛУ операнды представляются в последовательном коде, а операции производятся последовательно во времени над их отдельными разрядами. В параллельных АЛУ операнды представляются параллельным кодом и операции совершаются параллельно во времени над всеми разрядами операндов.

По способу представления чисел различают АЛУ:

– для чисел с фиксированной точкой;

– для чисел с плавающей точкой;

– для десятичных чисел.

По характеру использования элементов и узлов АЛУ делятся на блочные и многофункциональные:

В блочном АЛУ операции над числами с фиксированной и плавающей точкой, десятичными числами и алфавитно-цифровыми полями выполняются в отдельных блоках, при этом повышается скорость работы, так как блоки могут параллельно выполнять соответствующие операции, но значительно возрастают затраты оборудования.

В многофункциональных АЛУ операции для всех форм представления чисел выполняются одними и теми же схемами, которые коммутируются нужным образом в зависимости от требуемого режима работы.

По своим функциям АЛУ является операционным блоком, выполняющим микрооперации, обеспечивающие прием из других устройств (например, памяти) операндов, их преобразование и выдачу результатов преобразования в другие устройства (таблица 4.1).

4.2 Описание АЛУ на языке VHDL

Хотя язык VHDL имеет встроенные операторы сложения (+) и вычитания (–), они работают только с целыми и действительными числами и физическими типами. В частности, они не работают с типами BIT_VECTOR и типом

STD_LOGIC_VECTOR стандарта IEEE. Для этих типов в стандартных пакетах определены специальные операторы

Таблица 4.1 – Список стандартных функций АЛУ

Функция	Описание функции
R=X+Y	Сложение X и Y
R=X+Y+Cl	Сложение X и Y с переносом
R=X – Y	Вычесть Y из X
R=X - Y - Cl	Вычесть Y из X с заемом
R=Y - X -	Вычесть X из Y
K=Y - X - Cl	Вычесть X из Y с заемом
R= - X	Арифметическое отрицание X
R= - Y	Арифметическое отрицание Y
R=Y+1	Инкремент Y
R=Y – 1	Декремент Y
R=PASS X	Результат равен операнду X
R=PASS Y	Результат равен операнду Y
R=0 (PASS 0)	Очистить результат
R=ABS X	Результат равен абсолютному значению X
R=X AND Y	Логическое И (AND) X и Y
R=X OR Y	Логическое ИЛИ (OR) X и Y
R=X XOR Y	Исключающее логическое ИЛИ (XOR) X и Y
R=NOT X	Логическое отрицание X
R=NOT Y	Логическое отрицание Y

В пакете IEEE_std_logic_arith определены два новых типа массивов – SIGNED и UNSIGNED – и набор функций сравнения для операндов типа INTEGER, SIGNED и UNSIGNED. В данном пакете определены операции сложения и вычитания для операндов тех же типов, а также для 1-разрядных операндов типа STD_LOGIC и STDJJLOGIC.

При большом числе перекрывающихся функций сложения и вычитания не столь очевидно, каким окажется тип результата сложения или вычитания. Если хотя бы один из операндов принадлежит типу SIGNED, то обычно результат будет типа SIGNED, в противном случае результат будет типа UNSIGNED. Но если результирующее значение присваивается сигналу или переменной типа STD_LOGIC_VECTOR, то результат типа SIGNED или UNSIGNED преобразуется к этому типу. Разрядность любого результата обычно равна разрядности самого длинного операнда. Однако, когда операнд типа UNSIGNED участвует в одной операции с операндом типа SIGNED или INTEGER, его разрядность увеличивается на 1 для размещения в нем знакового бита, равного 0, и только после этого устанавливается разрядность результата.

В листинге 4.1 приведена VHDL-программа сложения 8-разрядных операндов различного типа, иллюстрирующая эти правила. Первый результат S объявлен как 9-разрядное двоичное число в предположении, что разработчика интересует перенос, который может возникнуть при сложении 8-разрядных операндов A и B типа UNSIGNED. С помощью оператора конкатенации & операнды A и B расширяются так, чтобы функция сложения помещала бит переноса в старший разряд результата.

Листинг 4.1 – VHDL-программа сложения и вычитания 8-разрядных целых чисел различных типов

```
library IEEE;
use  IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity vadd is
  port (
    A, B: in UNSIGNED (7 downto 0);
    C:   in SIGNED (7 downto 0);
    D:   in STD_L0GIC_VECT0R (7 downto 0);
    S:   out UNSIGNED (8 downto 0);
    T:   out SIGNED (8 downto 0);
    U:   out SIGNED (7 downto 0);
    V:   out STD_L0GIC_VECT0R (8 downto 0)
    );
end vadd;

architecture vadd.arch of vadd is
begin
S <= ('0' & A) + ('0' & B);
T <= A + C;
U <= C + SIGNED(D);
V <= C - UNSIGNED(D);
end vadd_arch;
```

Следующий результат T также является 9-разрядным, так как функция сложения расширяет операнд A типа UNSIGNED при его сложении с операндом C типа SIGNED. В третьей операции сложения 8-разрядный операнд D типа STD_LOGIC_VECTOR преобразуется в операнд типа SIGNED и складывается с операндом C, так что в результате получается 8-разрядное двоичное число U типа SIGNED. В последнем операторе величина D преобразуется в операнд типа UNSIGNED, автоматически расширяется на один разряд и вычитается из C, так что результат V оказывается 9-разрядным.

Так как сложение и вычитание являются довольно дорогими операциями в смысле числа требуемых вентилей, многие VHDL-средства синтеза будут пытаться многократно использовать блоки сумматора всякий раз, когда это возможно. В листинге 4.2, например, приведена VHDL-программа, включающая два различных сложения. Вместо того чтобы образовать два

сумматора и с помощью мультиплексора выбирать выход одного из них, синтезирующая программа может создать только один сумматор и с помощью мультиплексоров переключать его входы, в результате чего схема в целом будет иметь меньшие размеры.

Листинг 4.2 – VHDL-программа с многократным использованием сумматора

```
library IEEE;
use IEEE.std_logic-1164.all;
use IEEE.std_logic_arith.all;

entity vaddshr is
port (
    A, B, C, D: in SIGNED (7 downto 0);
    SEL: in STD_LOGIC;
    S: out SIGNED (7 downto 0));
end vaddshr;

architecture vaddshr_arch of vaddshr is
begin
S <= A + B when SEL = '1' else C + D;
end vaddshr_arch;
```

В библиотеке IEEE std_logic_arith имеются функции умножения для классов SIGNED и UNSIGNED, и эти функции представлены оператором "*". Таким образом, в программе, приведенной в лист. 4.3, умножение чисел без знака можно осуществить в одну строку простым оператором присваивания.

Листинг 4.3 – поведенческая VHDL-программа для комбинационного умножителя 8x8

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity vmul8x8i is
    port (
        X: in UNSIGNED (7 downto 0);
        Y: in UNSIGNED (7 downto 0);
        P: out UNSIGNED (15 downto 0)
    );
end vmul8x8i;

architecture vmul8x8i_arch of vmul8x8i is
begin
    P <= X * Y;
end vmul8x8i_arch;
```

4.3 Приборы и оборудования, применяемые в работе

1 Комплект разработки для ПЛИС Spartan 3AN.

2 Персональный компьютер с USB портом.

3 Программное обеспечение Xilinx ISE Design Suite 10.1.

4.4 Порядок выполнения работы

1 Запустить среду разработки Xilinx ISE Design Suite 10.1.

2 Разработать программу арифметико-логического устройства в соответствии с выданным вариантом задания.

3 Скомпилировать программу, провести исправление ошибок.

4 Выполнить тестирование проекта.

5 Подготовить комплект отладки и ПК. Подключить плату к компьютеру по интерфейсу USB. Подать питание на плату с помощью 9В блока питания. Установить связь МК с компьютером.

6 Записать программу во внутреннюю память ПЛИС.

7 Проверить работу микросхемы.

8 Запустить работу (выключить питание платы, отсоединить плату, выключить компьютер).

Варианты заданий по выполнению лабораторной работы представлены в таблице 4.2.

Таблица 4.2 – Варианты заданий

Номер варианта	Тип арифметико-логического устройства
1	8-командное АЛУ
2	12-командное АЛУ
3	14-командное АЛУ
4	16-командное АЛУ

4.5 Содержание отчета

1 Цель работы.

2 Листинг программы арифметико-логического устройства.

3 Окно моделирования работы программы.

4 Выводы.

4.6 Контрольные вопросы

1 Каково назначение и основные функции АЛУ?

2 Какова классификация АЛУ?

3 Каковы особенности использования типов операндов SIGNED и UNSIGNED?

4 Как возможно минимизировать схему с помощью мультиплексирования?

5 Объясните структуру объявления АЛУ.

6 Поясните особенности архитектуры разработанного АЛУ.

ЛИТЕРАТУРА

1 Уэйкерли, Дж. Проектирование цифровых устройств. Ч. 1 / Дж. Уэйкерли. – СПб. : Постмаркет, 2002 – 521 с.

2 Зотов, В. Ю. Проектирование встраиваемых микропроцессорных систем на основе ПЛИС фирмы Xillinx в САПР WebPack ISE / В. Ю. Зотов. – М. : Горячая линия-Телеком, 2006. – 520 с.

3 Тарасов, И. Е. Разработка цифровых устройств на основе ПЛИС Xillinx с применением языка VHDL / И. Е. Тарасов. – М. : Горячая линия-Телеком, 2005. – 252 с.

4 Кузелин, М. О. Осовременные семейства ПЛИС фирмы Xillinx : справ. пособие / М. О. Кузелин, Д. А. Кнышев, В. Ю. Зотов. – М. : Горячая линия-Телеком, 2004. – 440 с.

Св. план 2011, поз. 9

Учебное издание

Кракасевич Сергей Викторович
Давыдов Максим Викторович
Смирнов Александр Владимирович и др.

ПРОЕКТИРОВАНИЕ НА ОСНОВЕ МИКРОКОНТРОЛЛЕРОВ

Лабораторный практикум
для студентов специальности
«Медицинская электроника»
дневной и заочной форм обучения

В 2-х частях

Часть 1

Редактор Н. В. Гриневич
Корректор Е. Н. Батурчик

Подписано в печать 01.02.2012. Формат 60x84 1/16. Бумага офсетная.
Гарнитура «Таймс». Отпечатано на ризографе. Усл. печ. л. 3,14.
Уч.-изд. л. 3,0. Тираж 100 экз. Заказ 176.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
ЛИ № 02330/0494371 от 16.03. 2009. ЛП № 02330/0494175 от 03.04.2009.
220013, Минск, П. Бровки, 6