МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
Юго-Западный государственный университет
(ЮЗГУ)

Кафедра вычислительной техники

УТВЕРЖДАЮ
Проректор по учебной работе
О.Г.Локтионова
_____ 2017 г.

Проектирование бортовых электронных средств и интерфейсов

Методические рекомендации по выполнению лабораторных работ №1-4 по курсу «Проектирование бортовых электронных средств и интерфейсов» для студентов направления подготовки 09.03.01

Курск 2017

УДК 621.(076.1)

Составители: С.А. Дюбрюкс

Рецензент
Кандидат технических наук, доцент Ю.А. Халин

Проектирование бортовых электронных средств и интерфейсов: методические рекомендации по выполнению лабораторных работ №1-4 по дисциплине «Проектирование бортовых электронных средств и интерфейсов» / Юго-Зап. гос. ун-т; сост.: С.А. Дюбрюкс.- Курск, 2017. 43 с.: ил. 30.

Методические рекомендации содержат задания и примеры решения типовых задач, встречающихся при разработке комбинационных схем для обеспечения функционирования ряда интерфейсов. В лабораторной работе №1 рассматривается описание основных конструкций языка VHDL, приводится определение и назначение бита чётности, способ его вычисления при обработке последовательного кода. В лабораторной работе №2 рассматриваются назначение и особенности формирования CRC-последовательностей. В лабораторной работе №3 приводится определение бит-стаффинга, способы его аппаратной реализации через построение цифровых автоматов и реализацию их на языке VHDL. В лабораторной работе №4 рассматривается аппаратно-ориентированный способ реализации процедуры восстановления прошедших битстаффинг информационных последовательностей. Предназначены для студентов направления подготовки 09.03.01 очной и заочной форм обучения «Информатика и вычислительная техника».

Текст печатается в авторской редакции
Подписано в печать 10.11.17. Формат 60x84 1/16.
Усл. печ. л. Уч. – изд.л. Тираж 30 экз. Заказ 849 Бесплатно.
Юго-Западный государственный университет
305040, Курск, ул. 50 лет Октября, 94.

Содержание

Введение

Лабораторные работы из данного пособия посвящены практическому применению знаний основ языка VHDL на примере решения типовых задач, встречающихся при разработке комбинационных схем для обеспечения функционирования ряда интерфейсов, применяющихся в бортовом оборудовании. Они развивают у студентов навыки аппаратно-ориентированного программирования на языках описания цифровой аппаратуры и способствуют закреплению навыков, усвоенных при изучении курса дисциплины “Проектирование бортовых электронных средств и интерфейсов”.

В теоретической части изложен смысл основных конструкций языка VHDL, определение и назначение бита чётности, способ его вычисления при обработке последовательного кода. Описано назначение и особенности формирования CRC-последовательностей, определение бит-стаффинга, способы его аппаратной реализации через построение цифровых автоматов и реализацию их на языке VHDL. Приведён аппаратно-ориентированный способ реализации процедуры восстановления прошедших битстаффинг информационных последовательностей.

Цель работы: закрепление навыков проектирования, отладки и тестирования в САПР Active-HDL 8.3 на примере решения типовых

задач, встречающихся при разработке комбинационных схем для обеспечения функционирования ряда интерфейсов, применяющихся в бортовом оборудовании.

Выполнению работы предшествует опрос по теории работы и устное собеседование по методике ее выполнения.

Каждая работа оформляется студентом в виде отчета, который обязательно включает раздел, где анализируется и объясняется вся полученная информация.

Итогом работы является ее защита. Защита проводится устно, но обязательно индивидуально.

Основные теоретические сведения.

К лабораторной работе №1.

Контрольный бит чётности (паритета) применяется при последовательной передаче данных для контроля информации. Этот бит дополняет количество единичных бит данных до нечётного (или чётного в зависимости от используемого соглашения). Приём данных с неверным значением контрольного бита приводит к фиксации ошибки. Применительно к данной лабораторной работе под битом чётности подразумевается признак того, что поступающее на вход компонента двойное информационное слово содержит чётное число единиц.

PACKAGE (не обязательная часть)
ENTITY (описание входов/выходов пректируемого устройства)
ARCHITECTURE (описание функционирования устройства)

Параллельная обработка

- блоки
- подстановка компонентов
- присвоение сигналам значений логического уровня
- вызовы подпрограмм
- процессы

Последовательная обработка

- объявления переменных
- присвоение значений сигналам и переменным
- вызовы процедур
- if, case, loop, next, exit, return
- wait

ENTITY

Объявление элемента содержит в себе его имя и описание интерфейса, необходимое для его включения его в иерархию более высокого уровня. Это как бы взгляд на элемент снаружи. Формат объявления элемента следующий:

```
entity
is
        [generic_объявление настраиваемых параметров]
        [port_объявление портов]
end;
```

Порты в данном случае можно рассматривать как контакты элемента при включении его сгенерированного условно-графического обозначения в схему.

Порты должны быть сигналами типа STD_LOGIC, BIT, STD_LOGIC_VECTOR, BIT_VECTOR или INTEGER. Они не могут быть константами или переменными или иметь какой-либо другой (в том числе определяемый пользователем) тип. Каждый порт имеет, кроме того, “режим”,

который определяет направление прохождения через него сигнала. Существующие режимы: IN, OUT, INOUT или BUFFER.

С точки зрения внутреннего описания элемента:

IN - данные можно только прочесть, записать (присвоить значение) нельзя

OUT - данными можно только управлять (присвоить значение)

INOUT- можно делать и то, и другое

BUFFER-значение можно и прочесть, и присвоить. В отличие от IN, внешние по отношению к элементу устройства не могут управлять таким портом. Такой порт можно рассматривать как “непрямой” выход, состояние которого Вы можете прочесть, он управляет выходным буфером и внутренней логикой. Используется чаще всего для организации регистровой обратной связи.

Портам могут быть присвоены начальные значения, для инициализации перед началом моделирования. Исключение составляет IN - в том случае, если он подключен в внешним устройствам. Если значение по умолчанию не задано, порт устанавливается в состояние “X”. Следует учитывать, что все присвоенные при объявлении портам значения средствами синтеза игнорируются.

Итак, порты - это сигналы, которые видны снаружи. Они используются внутри поведенческого описания как сигналы, обращение к ним происходит по именам (и индексам, в случае необходимости).

Зарезервированное слово *generic* в объявлении элемента используется для объявления параметров, которые могут изменятся при использовании включений описываемого устройства в иерархию более высокого уровня. К числу таких параметров чаще всего относится размерность шин, их применение позволяет многократно использовать одно и то же описание. Настраиваемые параметры используются для создания параметризованных описаний, пример одного из которых приведен ниже.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity reg_bank is
  generic(width: positive:=32);
  port(reset, ck : in std_logic;
       d         : in std_logic_vector(width-1 downto 0);
       q         : out std_logic_vector(width-1 downto 0));
end;
architecture behavior of reg_bank is
begin
process (reset,ck,d)
  begin
    if reset = '1' then
      q <= (others=>'0');
    elsif (ck = '1' and ck'event) then
q <= d;
    end if;
  end process;
```

end;

Фактически, использование такого формы представления портов элементов позволяет в данном случае описать регистр, входные и выходные шины данных могут иметь различную размерность. При синтезе такие объявления рассматриваются как константы.

ARCHITECTURE

Архитектурное тело элемента описывает его поведение (функционирование). Каждое архитектурное тело ассоциируется по имени с одним объявлением элемента. В свою очередь, каждое объявление имеет соответствующее ему архитектурное тело.

Каждое архитектуртурное тело состоит из двух частей: описательной и набора операторов. В описательной части содержится объявление констант, компонентов, сигналов и т.д. Все эти объекты доступны только внутри данного архитектурного тела. Набор операторов содержит операторы BLOCK, PROCESS, вызовы параллельно выполняющихся процедур, присвоения сигналам значений, подстановку компонентов. За исключением содержимого оператора BLOCK, который представляет собой набор различных параллельно выполняемых операторов и сам по себе не является выполнимым, все остальные операторы в архитектурном теле выполняются одновременно. Формат архитектурного тела следующий:

ARCHITECTURE *[наименование архитектурного тела]* OF *[имя объявления]* IS
    [объявление констант]
    [объявление сигналов]
    [объявление типов]
    [объявление компонентов]
    [объявление атрибутов]
    [спецификация атрибутов]
BEGIN
    [парралельно выполняющиеся операторы]
END *[наименование архитектурного тела]*;

Сигналы соединяют между собой отдельные части (параллельно выполняющиеся выражения) архитектурного тела между собой и, через интерфейсные порты, с окружаещей элемент частью проекта в целом. Каждое параллельно выполняемой выражение в архитектурном теле представляет собой чтение текущего состояния сигнала, его обработку и присвоение сигналу нового значения логического уровня.

Принципиально важным моментом про разработке описаний является понятие процесса (оператор PROCESS). Процессы - основные “кирпичики”, из которых строиться любое архитектурное тело. Они описывают алгоритм функционирования последовательного устройства. Процесс представляет собой составной оператор, выполняющийся параллельно с остальными входящими в архитектуру операторами и содержащий набор последовательно выполняющихся операторов. Это означает, что если в архитектуру входят несколько процессов, то

они выполняются все одновременно, но при этом операторы, входящие в каждый конкретный процесс, выполняются внутри него последовательно, в порядке их записи в тексте.

Объекты

Объекты можно представить себе как именованные единицы хранения данных при моделировании и синтезе. Каждый такой объект определяется именем, типом и классом. Имя объекта представляет собой текстовую строку, определяемую пользователем, тип показывает, какие данные там могут храниться. Класс объекта, определяет набор допустимых действий с этим объектом. Существуют следующие классы:

⇒CONSTANT - константа имеет значение, которое присваивается ей при объявлении и не может изменяться. Константы могут быть любого поддерживаемого типа.

⇒VARIABLE - переменная имеет значение, которое может изменяться путем присвоения ей другого значения. Переменные могут быть любого поддерживаемого типа.

⇒SIGNAL - сигналы можно представить как проводники (или наборы проводников - шины) , логическое состояние которых определяется выражением присваивания значения сигналу. Сигнал может представлять собой, например, выход цепей комбинационной логики, выход триггера или регистра.

Переприсвоние значения сигнала внутри определения параллельно выполняющихся операций недопустимо. Другими словами, каждый сигнал (провод) может быть одновременно подключен только к одному задающему генератору.

Имя объекта может включать в себя цифры, буквы латинского алфавита и знак подчеркивания. Оно должно начинаться с буквы, может иметь в своем составе только один знак подчеркивания и не должно им заканчиваться. Имя объекта не может представлять собой зарезервированное слово (их список приведен в приложении 3)

В индексируемых объектах (например, массивах) индекс является целым числом, как представлено в приведенных ниже примерах

vector(0 to 3)

vector(3 downto 0)

Использование в качестве индексов переменных и сигналов не поддерживается.

Объекты могут объявляться в разных местах (в архитектуре, портах, процессах, подпрогаммах). Главное - чтобы они были объявлены до их использования. Единственным исключением из этого правила являются параметры цикла LOOP - их объявление и присвоение им значений являются частью оператора цикла.

Объявление констант

В одном выражении могут быть объявлены сразу несколько констант. Присваиваемое константе значение должно иметь тот же тип, что и константа. Формат записи объявления константы:

CONSTANT имя{,имя}:тип [(диапазон индексов)][:=начальное значение];

Если тип константы-неограниченный массив, диапазон индексов должен быть указан обязательно. Примеры объявления констант:

CONSTANT zero :INTEGER:=0;

CONSTANT zero_7 :std_logic_vector(6 downto 0):=”0000000”;

К константам можно обращаться просто по именам. Кроме этого, в константах типа массива можно обращаться к отдельному ее элементу, используя индекс или диапазон индексов для выделения части массива (4-х битный вектор в данном случае).

zero_7(3 downto 0)

Объявление переменных

В одном выражении могут быть объявлены сразу несколько переменных. Начальное присвоение значение переменной внутри процесса не поддерживается, и такое присвоение игнорируется средствами синтеза. Инициализация переменных поддерживается только внутри процедур и функций. Нерегистровые переменные внутри процесса должны быть проинициализированы до их использования. Значение переменной “0010” в нижеследующем примере игнорируется и последующие действия приводят к появлению ошибочных результатов.

```
process(s1,s2)
variable v1:std_logic_vector (0 to 3):=”0010”;—значение игнорируется
begin
        s<=v1;--ошибочный результат
end process;
```

Формат записи объявления переменной:

VARIABLE имя{,имя}:тип [(диапазон индексов)][:=начальное значение];

К переменным можно обращаться (или присваивать значение) по именам. Кроме этого, в переменных типа массива можно обращаться к отдельному ее элементу, используя индекс или диапазон индексов для выделения части массива.

Сигналы

В одном выражении могут быть объявлены сразу несколько сигналов. Инициализация сигналов при их объявлении игнорируется средствами синтеза.

Формат записи объявления сигнала:

SIGNAL имя{,имя}:тип [(диапазон индексов)];

К сигналам можно обращаться или присваивать значения как и переменным.

Пример объявления и использования сигналов:

```
SIGNAL clock, reset: std_logic;
SIGNAL q1, count:std_logic_vector (15 downto 0);
process
begin
        wait until clock='1';
        if reset ='1' then
        q1<="0000000000000000";
        else q<=count;
        end if;
end process;
```

Выражения языка VHDL

Логические операторы

VHDL поддерживает следующие логические операторы:

- and
- or
- nand
- nor
- xor
- xnor
- not

Эти операторы определены для типов bit, boolean, bit_vector и их модификаций. Они являются наиболее простыми и универсальными конструкциями языка. Если рассматривать ниже приведенных пример с точки зрения его схемной реализации, то использование записи логического оператора равносильно использованию в схеме вентиля, выполняющего соответствующую логическую функцию, а описание сигнала соответствует электрической связи (проводу) в схеме. Пример использования логических операторов приведен ниже.

```
library ieee;use ieee.std_logic_1164.all;use ieee.numeric_std.all;

entity eqs is  port (a1, a2, a3, a4, a5, a6,       b1, b2, b3, b5, b6 : std_logic;       y1, y2, y3, y4, y5, y6 : out std_logic);end eqs;

architecture example of eqs is  begin       y1 <= a1 and b1;          y2 <= a2 or b2;  y3 <= a3 xor b3;          y4       <=
not(a4);          y5 <= a5 nand b5;       y6 <= a6 nor b6;end example;
```

Операторы сравнения

=	равно
/=	не равно
>	больше чем
<	меньше чем
>=	больше или равно
<=	меньше или равно

Операторы равенства (= и /=) стандартно определены для всех типов данных. Операторы отношения (>=, <=, >,<) определены для числовых типов,

перечисляемых и некоторых массивов. Переопределение всех этих операторов для модифицированных типов данных находится в соответствующих PACKAGE-блоках (вместе с описанием модифицированного типа). Их использование требует включения соответствующего PACKAGE-блока в текст проекта с использованием директивы USE.

Результат выполнения всех этих операций - булев.

Пример использования операторов отношения приведен ниже.

```
entity rel_ops is
    port (a,b:in bit_vector (0 to 3); m:out boolean);
end rel_ops;

architecture EXAMPLE of  rel_ops is
        begin
                m<=a=b;
        end EXAMPLE;
```

Следует отметить, что реализация операторов равенства требует значительно меньших ресурсов кристалла (задействованных вентилей) чем реализация операторов неравенства. Следующий пример требует для реализации вдвое больше ресурсов кристалла, чем предыдущий.

```
entity rel_ops is
    port (a,b:in bit_vector (0 to 3); m:out boolean);
end rel_ops;

architecture EXAMPLE of  rel_ops is
        begin
                m<=a>=b;
        end EXAMPLE;
```

Арифметические операторы

+	сложение
-	вычитание
*	умножение
/	деление
mod	по модулю
rem	остаток от деления
abs	абсолютное значение

Реализация операторов сложения и вычитания занимает достаточно большое количество ресурсов кристалла, а остальные операторы являются крайне ресурсоемкими. Пример использования этих операторов приведен ниже.

```
        library ieee;
        use ieee.std_logic_1164.all;
        use ieee.numeric_std.all;
        entity mult4 is
          port(data_a, data_b  : in std_logic_vector(3 downto 0);
               data_out        : out std_logic_vector(7 downto 0));
        end;
        architecture behavior of mult4 is
        begin
```

```
   data_out <= std_logic_vector(unsigned(data_a) rem unsigned(data_b));
end;
```

В этом примере следует обратить особое внимание на строчку:

use ieee.numeric_std.all

Включение ее в текст описания делает видимыми функции и модифицированные типы данных, описанные PACKAGE-блоке NUMERIC_STD. (см. приложение 2). Это делает возможным использование перегружаемых функций (rem - в данном примере), изначально определенных только для целых чисел и переопределенных для векторных типов данных. Запись *unsigned(data_a)* представляет собой преобразование данных типа std_logic_vector к типу unsigned. Тип unsigned является модифицированным типом, описанным в NUMERIC_STD и там же содержится определение перегружаемой функции *rem*, для операций с данными типа unsigned.

Преобразование данных приведено для примера. Описание

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mult4 is
  port(data_a, data_b  : in unsigned (3 downto 0);
       data_out        : out unsigned(7 downto 0));
end;
architecture behavior of mult4 is
begin
 data_out <= data_a rem data_b;
end
```

даст тот же результат.

Приоритет выполнения операций

Уровень приоритета	Вид операций	Операторы
1	унарные	abs, not
2	умножение/деление	*, /, mod, rem
3	сложение/вычитание	+, -, &
4	отношение	=, /=, <, <=, >, >=
5	логические	and, or, nand, nor, xor

Операторы (функции) сдвига и вращения (циклического сдвига)

В операциях (функциях) сдвига и вращения (циклическогосдвига) левый операнд должен представлять собой одномерный массив, кождый элемент которого имеет тип BIT или BOOLEAN (или модификацию этих типов), правый операнд должен быть константой, целым числом или выражением, результатом выполнения которого является целое число.

VHDLсодержит следующие операторы сдвига и вращения (циклического сдвига)

⇒SHIFT_LEFT(vector,n) - сдвиг влево на n бит, значения левых n бит пропадают, n правых бит устанавливаются в “0”.

⇒SHIFT_RIGHT(vector,n)- сдвиг вправо на n бит, значения правых n бит пропадают, n левых бит устанавливаются в “0”.

В качестве примера их использования ниже приведен пример сдвигового регистра:

```
library ieee;use ieee.std_logic_1164.all;use ieee.numeric_std.all;entity shiftl is port(clk : in std_logic;
-- тактовый вход ldst: in std_logic; -- управление режимом работы регистра s : in std_logic_vector (2 downto 0); -- задание количества бит, на которое нужно произ
-- вести сдвиг d : in std_logic_vector (7 downto 0); -- входные данные q : out std_logic_vector (7 downto 0)); -- выходные данныеend;
architecture behavior of shiftl is signal s_sig : unsigned (2 downto 0); signal d_sig : unsigned (7 downto 0); signal q_sig : unsigned (7 downto 0);begin -- преобразование типов s_sig <= unsigned(s); d_sig <= unsigned(d); q <= std_logic_vector(q_sig); -- описание регистра process (clk,ldst,s_sig,d_sig) begin if (clk='1' and clk'event) then if (ldst='1') then q_sig <= d_sig; elsif (s_sig=b"000") then q_sig <= shift_right(q_sig,0); elsif (s_sig=b"001") then q_sig <= shift_right(q_sig,1); elsif (s_sig=b"010") then q_sig <= shift_right(q_sig,2); elsif (s_sig=b"011") then q_sig <= shift_right(q_sig,3); elsif (s_sig=b"100") then q_sig <= shift_right(q_sig,4); elsif (s_sig=b"101") then q_sig <= shift_right(q_sig,5); elsif (s_sig=b"110") then q_sig <= shift_right(q_sig,6); elsif (s_sig=b"111") then q_sig <= shift_right(q_sig,7); end if; end if; end process;end;
```

⇒ ROTATE_LEFT(vector,n)- циклический сдвиг влево , значения правых n бит последовательно заменяются значениями левых n бит.

⇒ ROTATE_ RIGHT(vector,n) - циклический сдвиг вправо , значения левых n бит последовательно заменяются значениями правых n бит.

Следует учитывать то, что если правый операнд является отрицательным числом, то реализуется двунаправленный сдвиг, что при реализации приводит к дополнительным затратат ресурсов кристалла, так что такой ситуации следует избегать.

Параллельно выполняемые действия

Архитектура описания элемента на VHDL всегда представляет собой набор параллельно выполняемых операторов.

PROCESS

Это составной оператор, содержащий в себе набор последовательно выполняемых действий. Если в архитектуре содержится несколько процессов, то все они выполняются одновременно, в каждый момент времени внутри каждого процесса выполняется только один оператор.

Процесс связан со всей остальной архитектурой проекта через сигналы и порты, объявленные вне его. Формат записи следующий:

```
метка PROCESS список чувствительности
        объявление внутренних объектов
        BEGIN
        набор последовательно выполняющихся операций
        END;
```

Необязательная метка присваивает процессу имя. Список чувствительности (также необязательный) содержит перечень всех сигналов (включая порты), изменения логического состояния которых должно приводить к выполнению

процесса. Все средства синтеза проверяют полноту этого списка. Список чувствительности должен отсутствовать в том случае, если в процессе есть оператор WAIT.

Объявления внутренних объектов процесса может содержать в себе директиву включения USE, объявления и тела подпрограмм, объявления типов, сигналов, констант и переменных. Все эти объекты доступны только внутри этого процесса.

В принципе, можно считать что поведение процесса определяется последовательностью внутренних операций. После того, как выполнен последний последовательный оператор , выполнение процесса начинается снова. Исключение составляют процессы, имеющие список чувствительности: выполнение такого процесса начинается снова в случае изменения логического уровня одного сигналов, входящих в этот список. В том случае, если в состав процесса входит оператор WAIT, (и, соответственно, отсутствует список чувствительности) выполнение прцесса приостанавливается до тех пор, пока условие выполнения в этом операторе не примет значение TRUE. Результатом синтеза элемента, описываемого процессом, может быть как комбинационная (не синхронная), так и синхроная схема. Если в состав процесса входят WAIT или signal'event, то результатом синтеза будет синхронная схема. Из этого можно сделать вывод, что процесс имеет свой первоначальный смысл только при описании последовательных (синхронных) действий. Если описываемые действия имеют смысл параллельных, удобнее пользоваться другими конструкциями языка (блоками, параллельно выполняющимися операторами). В следуещем примере приведен процесс, результатом синтеза для которого является комбинационная схема (счетчик по модулю 10).

```
entity COUNTER is
        port (CLEAR: in BIT;
        IN_COUNT: in INTEGER range 0 to 9;
        OUT_COUNT: out INTEGER range 0 to 9);
        end COUNTER;
architecture EXAMPLE of COUNTER is
        begin
        process(IN_COUNT, CLEAR)
                begin
                if (CLEAR = '1' or IN_COUNT = 9) then
                OUT_COUNT <= 0;
                else
                OUT_COUNT <= IN_COUNT + 1;
                end if;
        end process;
end EXAMPLE;
```

Этот процесс чувствителен к изменению двух сигналов: CLEAR и IN_COUNT и управляет состоянием сигнала OUT_COUNT. Если CLEAR=1 и IN_COUNT=9, OUT_COUNT становиться равным 0, в другом случае значение OUT_COUNT становиться на 1 больше чем IN_COUNT.

Так как в этом процессе нет оператора WAIT, то результатом синтеза является комбинационная схема. Другим подходом к описанию счетчика является возврат значения счетчика в процесс с оператором WAIT:

```
entity COUNTER is
        port (CLEAR: in BIT;
        CLOCK: in BIT;
        COUNT: buffer INTEGER range 0 to 9);
        end COUNTER;
architecture EXAMPLE of COUNTER is
        begin
        process
                begin
                wait until CLOCK'event and CLOCK = '1';
                if (CLEAR = '1' or COUNT >= 9) then
                COUNT <= 0;
                else
                COUNT <= COUNT + 1;
                end if;
        end process;
end EXAMPLE;
```

Если на CLOCK-передний фронт, начинается выполнение оператора IF. В том случае, если значение счетчика не должно быть обнулено, оно увеличивается на 1. Синтезированная по этому описанию схема имеет в своем составе 4 FF-триггера, предназначенных для хранения значения переменной COUNT. Триггера необходимы, так как значение COUNT может быть прочитано до его установки, следовательно оно должно схраняться с предыдущего такта.

Если в ходе выполнения процесса сигналу присваивается значение логического уровня, то процесс является “управляющим” по отношению к этому сигналу. Если больше чем один процесс или другой параллельно выполняющийся оператор управляют одним сигналом, то принят термин “multiply drivers”. Это самая распространенная ошибка в описаниях, поэтому название термина приведено в оригинале. Сигнал может иметь несколько задающих процессов только в случае использования буферов с тремя состояниями. Пример приведен ниже:

```
A_OUT <= A when ENABLE_A else 'Z';
B_OUT <= B when ENABLE_B else 'Z';
process(A_OUT)
        begin
        SIG <= A_OUT;
        end process;
        process(B_OUT)
        begin
        SIG <= B_OUT;
end process;
```

BLOCK

Это составной оператор, представляющий собой проименованный набор параллельно выполняемых действий, локально определенных констант, типов, сигналов, подпрограмм и компонентов.

Оператор BLOCK присваивает имя группе параллельно выполняемых операторов. Он используется для организации внутри описания элемента иерархии. Это бывает полезно в случае больших проектов, так как делает описание более читаемым и хорошо структурированным. Формат записи оператора:

метка: block
{ объявление внутренних объектов }
begin
{ параллельно выполняющиеся операторы }
end block [метка];

Метка является обязательной: она присваивает блоку имя.

К лабораторной работе №2.

Вычисление CRC-кода — это процесс преобразования целого потока данных в одно короткое «проверочное слово», удобное тем, что является очень «чувствительным» к малейшим изменениям в этом потоке данных. Наиболее часто в технике встречаются две разновидности CRC кодов — CRC-16 и CRC-32, которые являются 16- и 32-разрядными проверочными словами соответственно.

Алгоритм вычисления CRC представляет весь поток данных как одно огромное число и делит его на другое, выбранное число, называемое «образующим полиномом». Частное от этого деления игнорируется, а остаток как раз и является искомым проверочным словом. Образующие полиномы являются стандартными для различных протоколов (рисунок 1). Их можно задавать в двоичных кодах, где единицам соответствуют единичные коэффициенты полиномов. Например коду CRC-4 из примера будет соответствовать код 1101.

Код	Порождающий многочлен $g(X)$
CRC-4	$1 + X + X^4$
	Используется в ISDN
CRC-8	$(1 + X)(1 + X^2 + X^3 + X^4 + X^5 + X^6 + X^7) = 1 + X + X^2 + X^8$
	Используется в ATM в качестве HEC
CRC-12	$(1 + X)(1 + X^2 + X^{11}) = 1 + X + X^2 + X^3 + X^{11} + X^{12}$
CRC-16	$(1 + X)(1 + X^+ X^{15}) = 1 + X^2 + X^{15} + X^{16}$
	IBM
CRC-16	$(1 + X)(1 + X + X^2 + X^3 + X^4 + X^{12} + X^{13} + X^{14} + X^{15}) = 1 + X^5 + X^{12} + X^{16}$
	Является стандартом CCITT для HDLC и LAPD
CRC-32	$1 + X + X^2 + X^4 + X^5 + X^7 + X^8 + X^{10} + X^{11} + X^{12} + X^{16} + X^{22} + X^{23} + X^{26} + X^{32}$
	Используется в HDLC

Рисунок 1 – Наиболее распространённые образующие полиномы

Действия передатчика при формировании защищённых кодов следующие:

1. Выбор полинома P, в результате автоматически становится известной его степень N.

2. Добавление к исходной двоичной S последовательности N нулевых битов. Это добавление делается для гарантированной обработки всех битов исходной последовательности.

3. Выполнение деления дополненной N нулями исходной строки S на полином P по правилам двоичной арифметики (рисунок 2) пока не выполнится одно из условий:

- частичный остаток не станет меньше полинома и все оставшиеся разряды будут равны нулю, в этом случае данный остаток и составит CRC;
- частичный остаток не станет равен нулю и все оставшиеся разряды будут равны нулю, в этом случае данный CRC=0;
- пока в добавленных разрядах не появится хотя бы одна единица, CRC будет равен содержимому добавленных разрядов.

4. Формирование окончательного сообщения, которое будет состоять из двух частей: собственно сообщения и добавленного в его конец значения CRC.

К примеру, вычисление по этому алгоритму CRC для исходной последовательности 11010011100010110100 и сама окончательная последовательность на стороне источника будут выглядеть так, как показано на рис. 2. Из рисунка 3 видно, что в начале вычисления исходная последовательность 11010011100010110100 дополняется нулями в количестве, равном степени полинома (P=1011 - степень поли-

нома N=3): 11010011100010110100+000. При выполнении деления эти дополнительные биты гарантируют, что все биты исходной последовательности примут участие в процессе формирования значения CRC.

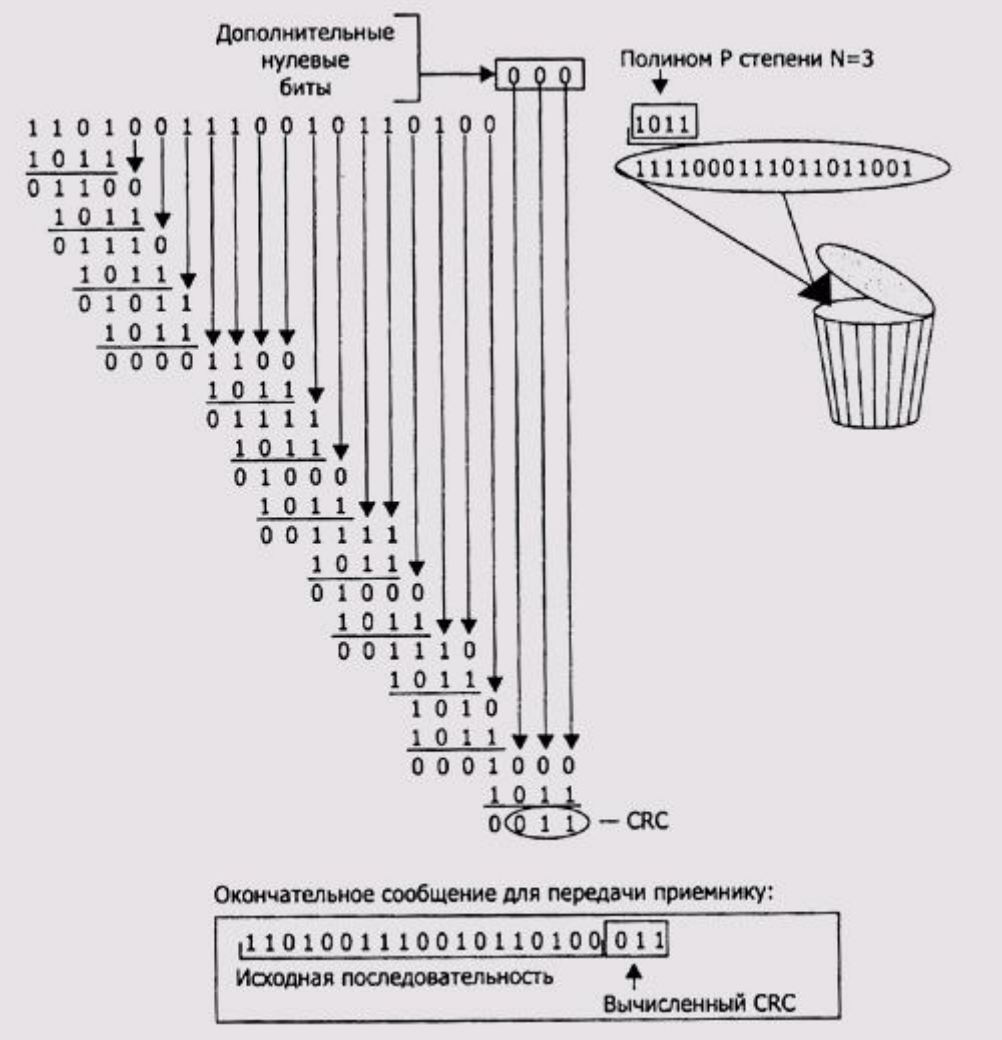

Рисунок 2 - Схема формирования выходного сообщения из исходного с использованием прямого CRC-алгоритма

Результирующая последовательность получается равной исходной последовательности, дополненной значением CRC: 11010011100010110100+011. Длина присоединяемого к исходной последовательности значения CRC должна быть равна степени полинома, даже если CRC, как в нашем случае, имеет ведущие нули. Это очень важный момент, понимание которого является ключом к пониманию сути процессов, происходящих на стороне приемника при получении и определении целостности исходного сообщения. Действия алгоритма для приемника просты — выполнить деление полученной последовательности на полином. При этом для выполнения деления нет необходимости дополнять исходную последовательность нулями, тем более что на практике соблюдение этого условия крайне неудобно. Приемник попросту выполняет CRC-деление полученной исходной строки (дополненной в конце исходным значением CRC) на полином и анализирует остаток. Если

остаток от этого деления нулевой, то исходная последовательность не была нарушена во время передачи. В обратном случае существует очень большая вероятность нарушения целостности исходной последовательности, и нужно принимать дополнительные меры по выяснению и исправлению ситуации. Одной из таких мер может быть попытка восстановления нужного значения CRC.

Описанный выше алгоритм (рис.3) вычисления значения CRC называется прямым и является наиболее удобным для аппаратной реализации.

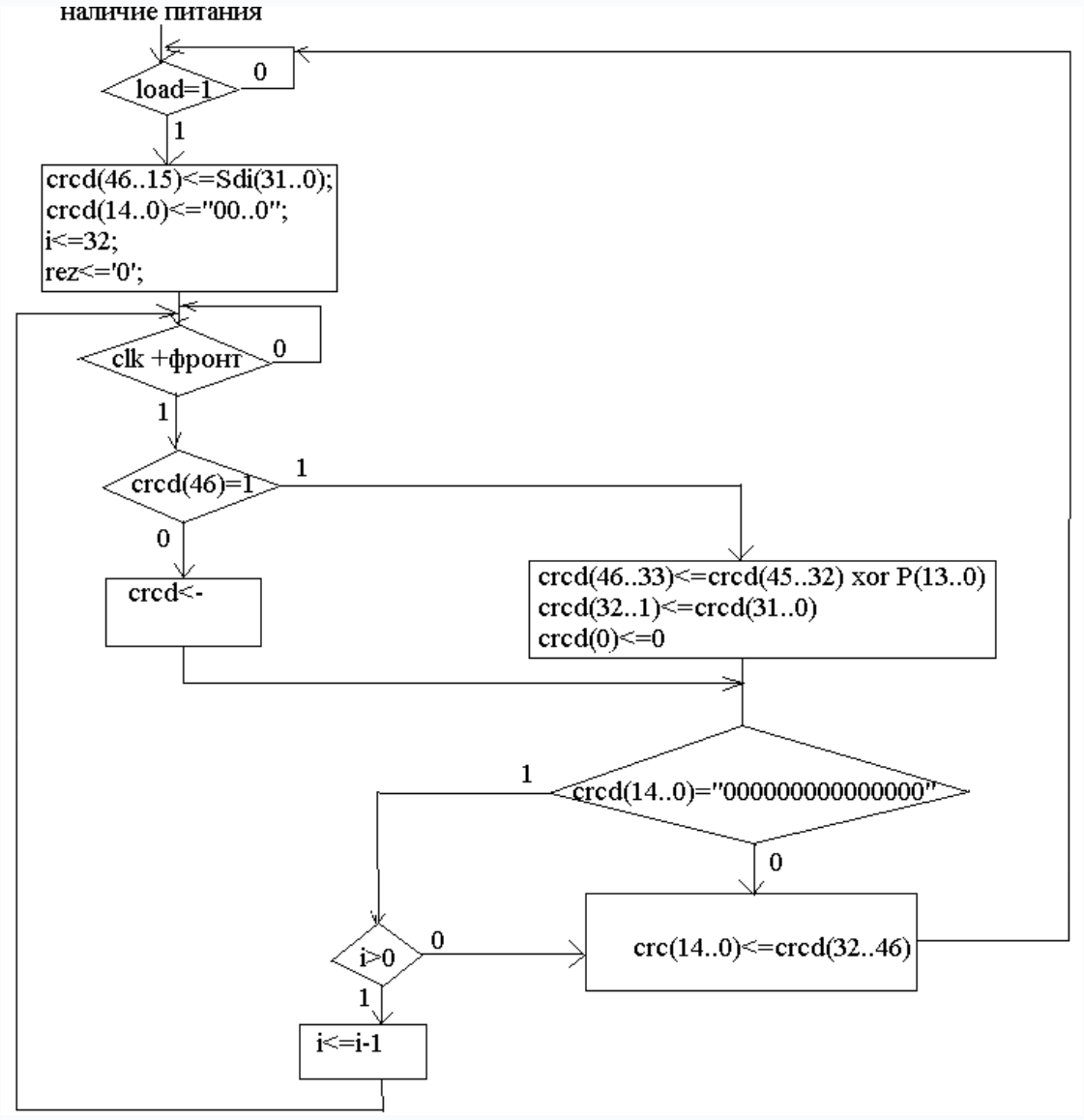

Рисунок 3 – аппаратно-ориентированный прямой CRC-алгоритм, использующий сигналы рисунка 1

К лабораторным работам №3-4.

Данные, передаваемые в форме сообщения в последовательных интерфейсах, обычно снабжаются *заголовком* и *концевиком,* в которых содержится информация, необходимая для обработки сообщения на соответствующем уровне: указатели ти-

па сообщения, адреса отправителя, получателя, канала, порта и т. д. Заголовок и концевик называются *обрамлением сообщения* (данных).

Средства управления нижнего уровня оперируют с данными, указанными в обрамлении, как с данными на конверте. При передаче сообщения на вышестоящий уровень сообщение “освобождается от конверта”, в результате чего на следующем уровне обрабатывается очередной “конверт”. Таким образом, каждый уровень управления оперирует не с самими сообщениями, а только с “конвертами”, в которых “упакованы” сообщения. Поэтому состав сообщений, формируемых на верхних уровнях, никак не влияет на функционирование нижних уровней управления передачей.

На нижнем, физическом, уровне в качестве заголовка и концевика используются специальные коды, например байт 01111110, который может встретиться в данных более, высокого уровня. Если не принять специальных мер, то последовательность битов 01111110 в данных будет ошибочно воспринята аппаратурой передачи как заголовок сообщения. Для исключения этого используется процедура обеспечения *прозрачности канала* — **битстаффинг.** То есть битстаффинг включает в себя поиск и исключение из кода последовательности бит, совпадающей с флагом (заголовок и концевик) путём добавления в найденную последовательность “разбивающего” бита –(рис.1в, 1г). Данный процесс реализуется передатчиками интерфейсных контроллеров, приёмники выполняют обратную процедуру – поиск вновь образованной в результате битстаффинга последовательности и исключение из неё вставленного передатчиком бита.

Поиск флага в последовательном коде удобно осуществлять с помощью цифрового автомата Мили. В качестве источника входного сигнала берётся последовательный вход данных x, выходы последовательного кода sout признака комбинации pr зависят как от предыстории переходов, так и от текущих значений на входе x. Выход шины A отображает текущее состояние автомата. Сброс и установка автомата в начальное состояние A0 осуществляется по положительному уровню сигнала start (рис.1б). При обнаружении признака (pr=1) на выход мультиплексора и соответственно всего компонента A1 подаётся разбивающий бит bit (равный 0 или 1).

Важная деталь: по найденному признаку pr и частоте clk формируется новая частота nclk, которая инверсна clk и имеет “пробелы” (рис.1г) в моменты, когда признак pr=1. По данной частоте осуществляется сдвиг в регистре SR32CLE. “Пробел” необходим для того, чтобы при вставке разбивающего бита не происходил сдвиг данных в регистре и бит исходной последовательности передаваемых данных не терялся, а откладывался на 1 такт.

Лабораторная работа №1

Формирование бита чётности.

Цель работы: закрепление навыков проектирования, отладки и тестирования в САПР Active-HDL 8.3 на примере реализации схемы вычисления бита чётности информационной части кадра, а также приобретение навыков перевода схем, заданных графически, в VHDL-код.

Задание на лабораторную работу.

Создать и провести моделирование работы структурного блока (A2 на рисунке 1), выполняющего функцию формирования бита чётности для двойного слова, поступающего в параллельном коде на вход блока A2 по шине D в соответствии с временной диаграммой на рисунке 1.1. Среда моделирования ActiveHDL 8.3.

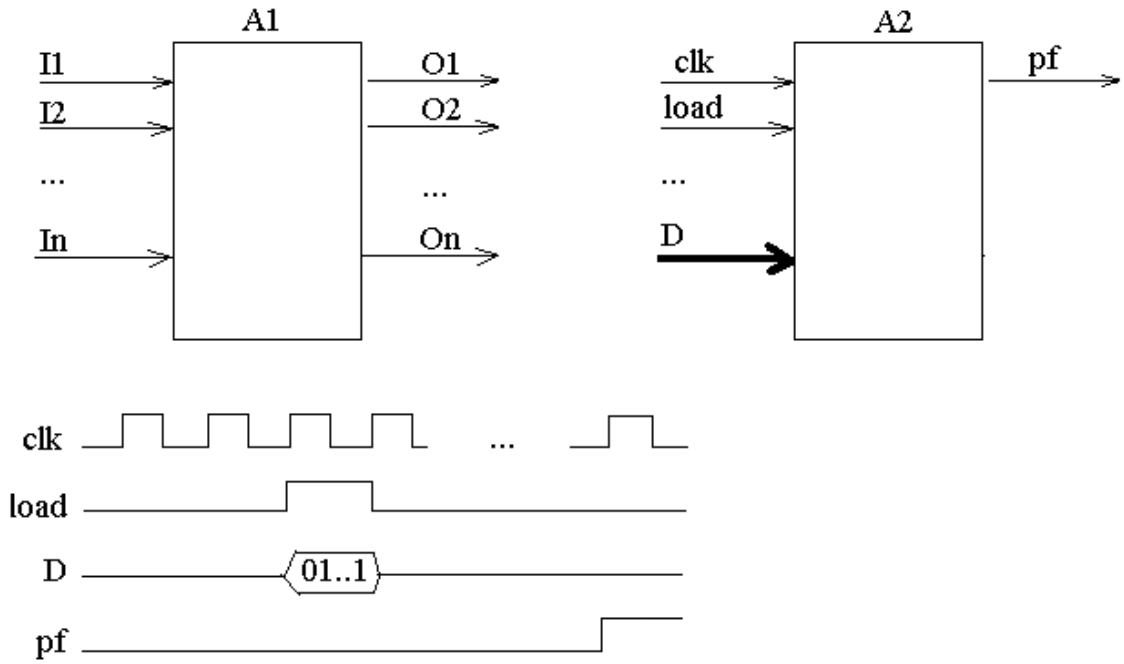

Рисунок 1.1 - структурный элемент для вычисления бита чётности.

Порядок выполнения работы:

1. Создайте проект в среде Active-HDL 8.3. Для этого запустите среду ActiveHDL 8.3, выбрав соответствующий значок. Появляется окно следующего вида (рисунок 1.2). Нажмите кнопку **Next**.

Рисунок 1.2 – Стартовое окно ActiveHDL 8.3.

2. В появившемся окне **Getting Started** (рис.1.3) выставите флажок **Create new workspace**, после чего нажмите кнопку **Next**.

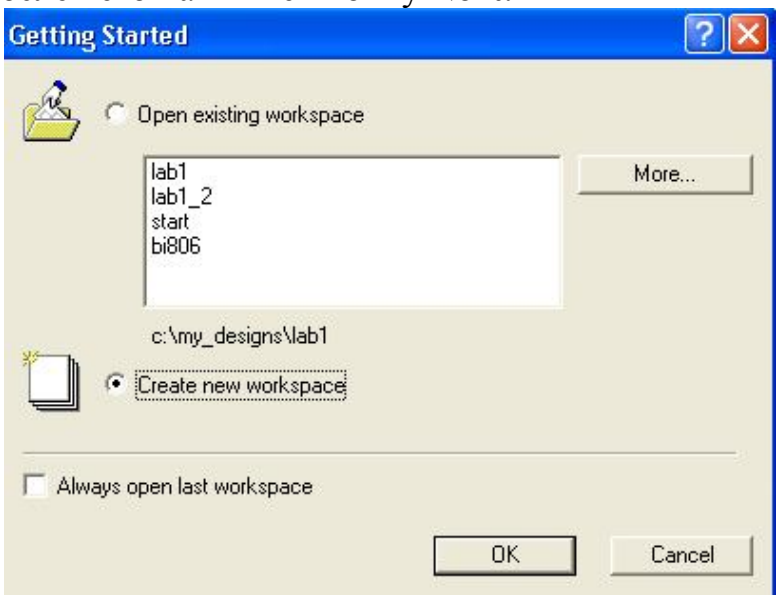

Рисунок 1.3 – окно выбора проекта.

3. В появившемся окне **New Workspace** (рис.1.4) в поле **Type the workspace name** в качестве имени файла введите фамилию (на английском). Выставьте флажок **Add New Design to Workspace**. Нажмите кнопку **OK**.

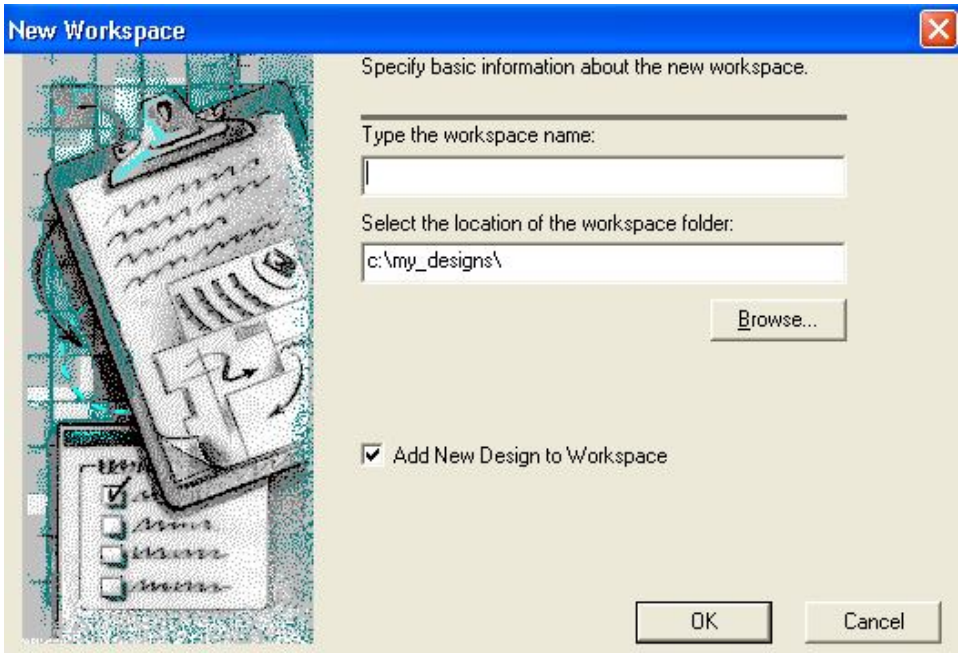

Рисунок 1.4 – ввод имени проекта.

4. В появившемся окне **New Design Wizard** выберите установки согласно рисунку 1.5, нажмите **Далее**.

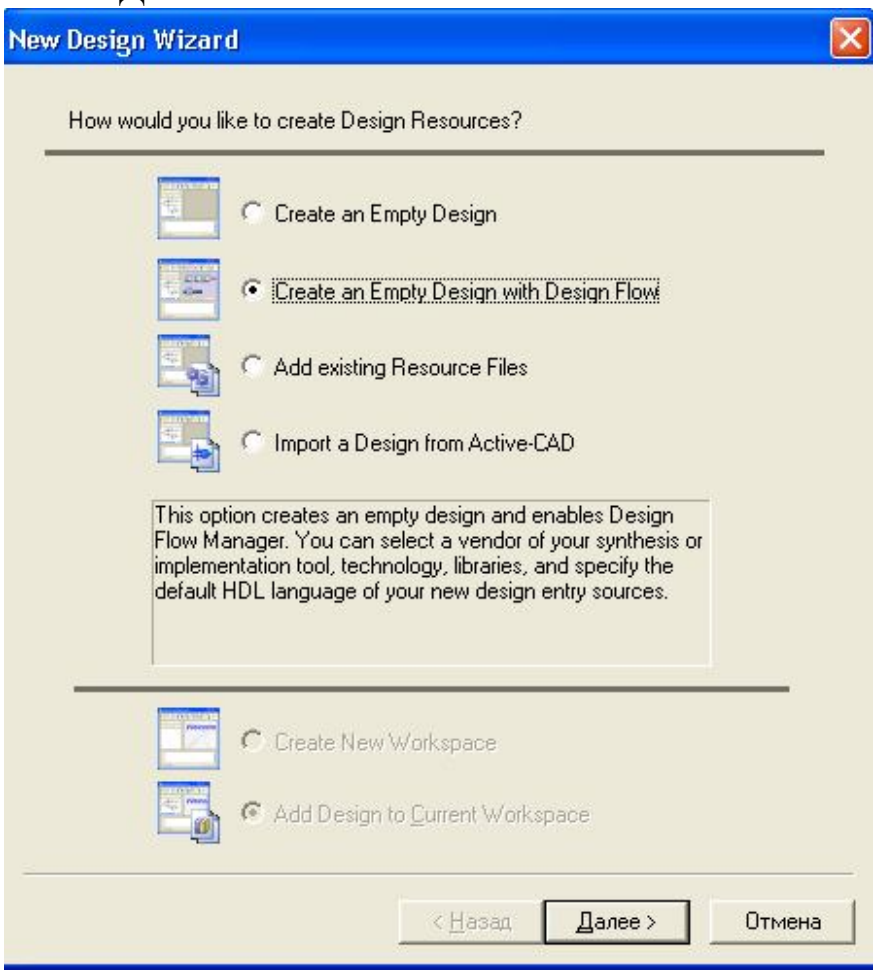

Рисунок 1.5 – выбор маршрута создания проекта.

5. В появившемся окне выставите установки в соответствии с рисунком 1.6, после чего нажмите **Далее**.

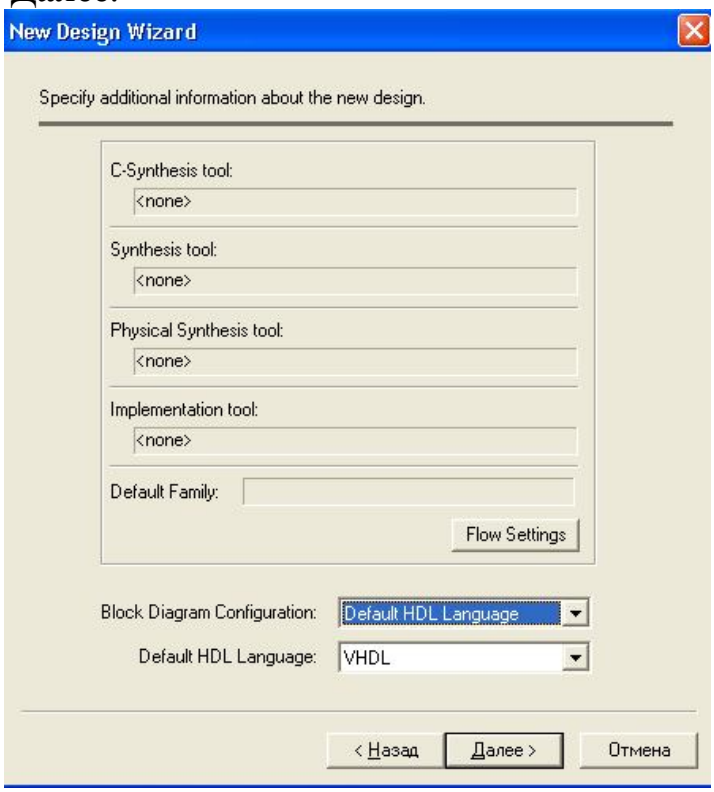

Рисунок 1.6 – выбор стиля создания проекта.

6. В появившемся окне (рис.1.7) в поле Type the design name введите имя проекта и нажмите **Далее**.

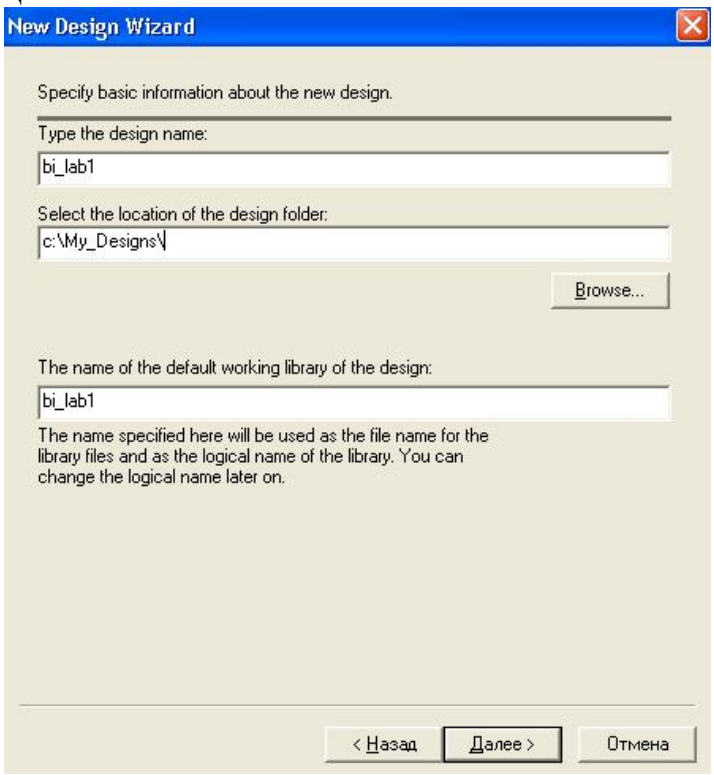

Рисунок 1.7 - ввод имени основного проектного файла.

7. В появившемся окне (рис.1.8) нажмите кнопку **Готово**. Создание заготовки проекта завершено.

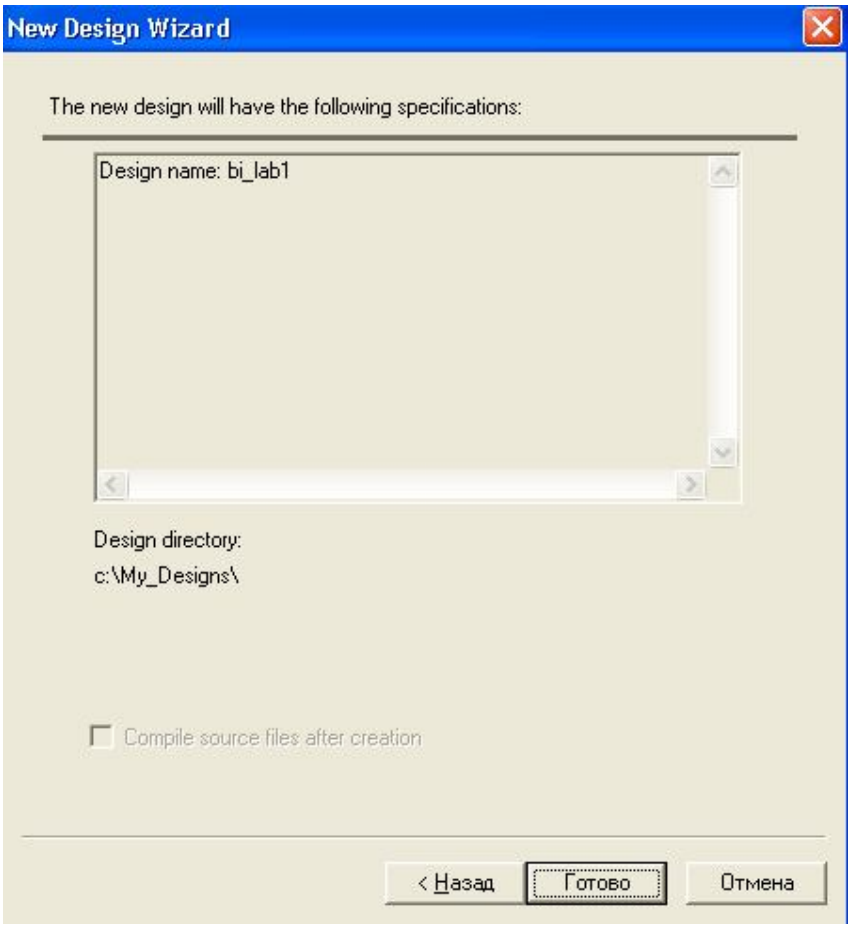

Рисунок 1.8 - окончание создания проекта.

8. В появившемся окне (рис.1.9) нажмите кнопку выберите редактор кода HDE.

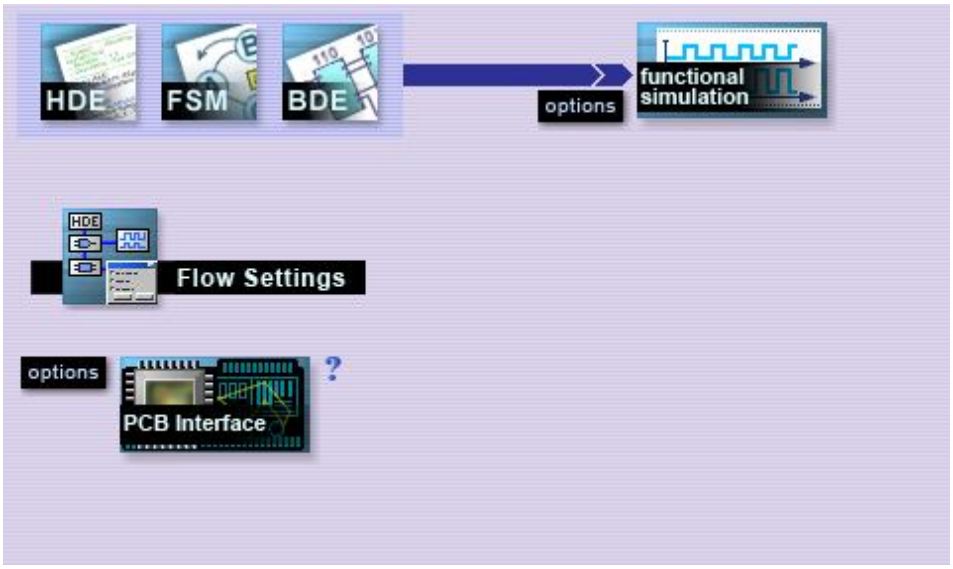

Рисунок 1.9 – выбор редактора кода.

9. В появившемся окне **HDL Editor** (рис.1.10) выберите флажок VHDL. Нажмите **ОК**.

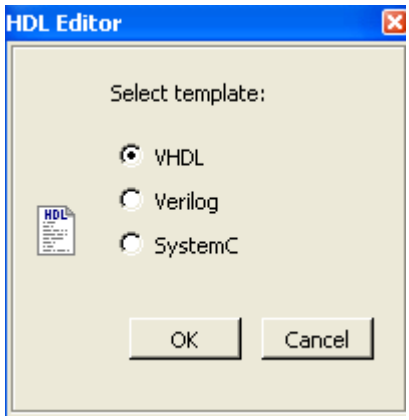

Рисунок 10 – выбор языка описания проекта.

10. В появившемся окне мастера создания кода (рис.1.11) выберите флажок Add the generated file to the design. Нажмите **Далее**.

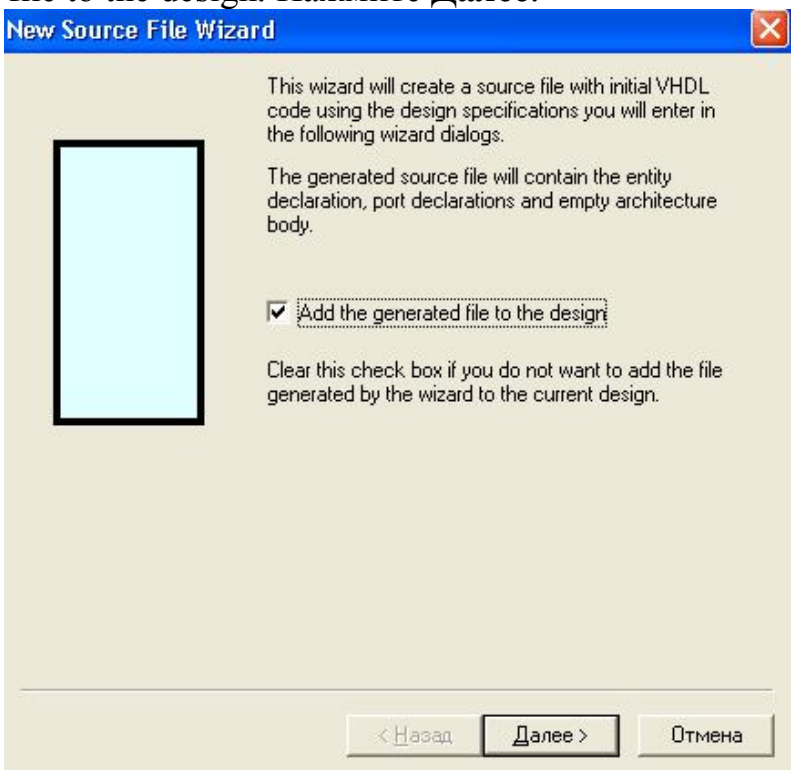

Рисунок 1.11 – создание головного файла проекта.

11. В появившемся окне (рис.1.12) введите имя файла кода. Нажмите **Далее**.

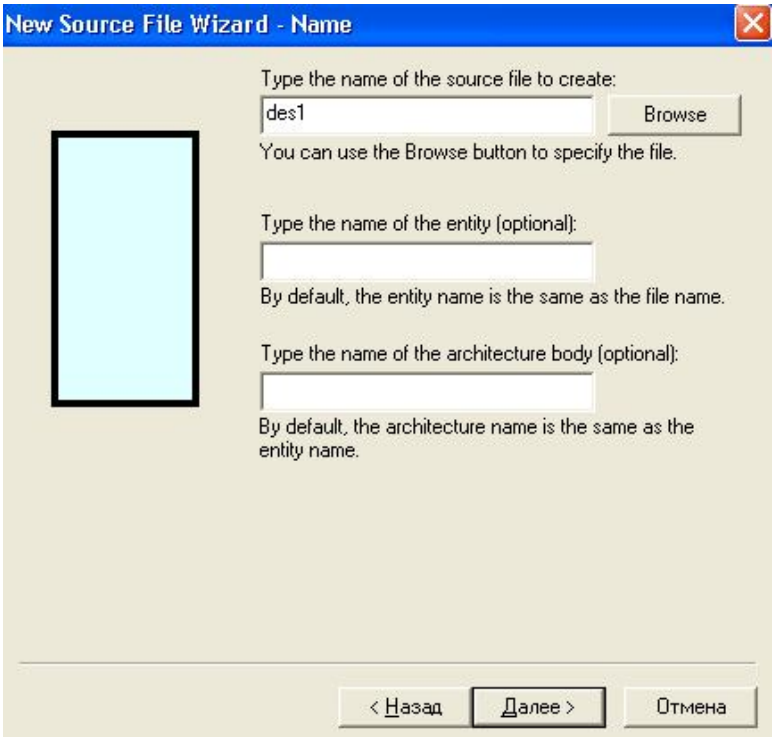

Рисунок 1.12 – ввод имени головного файла проекта

12. В появившемся окне выставьте установки портов как на рисунке 1.13. Нажмите **Готово**. Появляется окно редактора кода (рис.1.14).

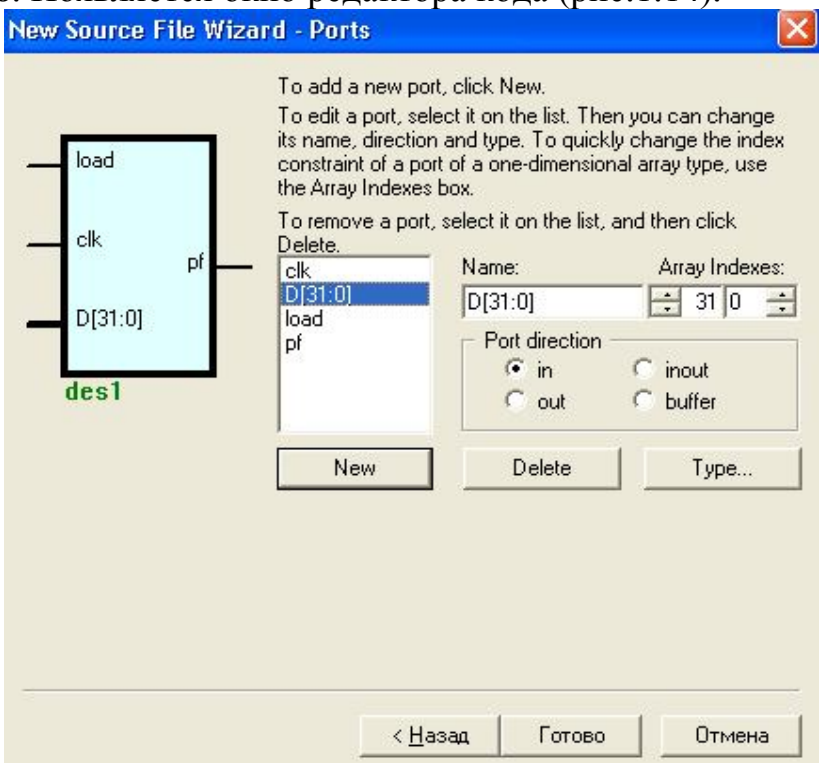

Рисунок 1.13 – ввод портов.

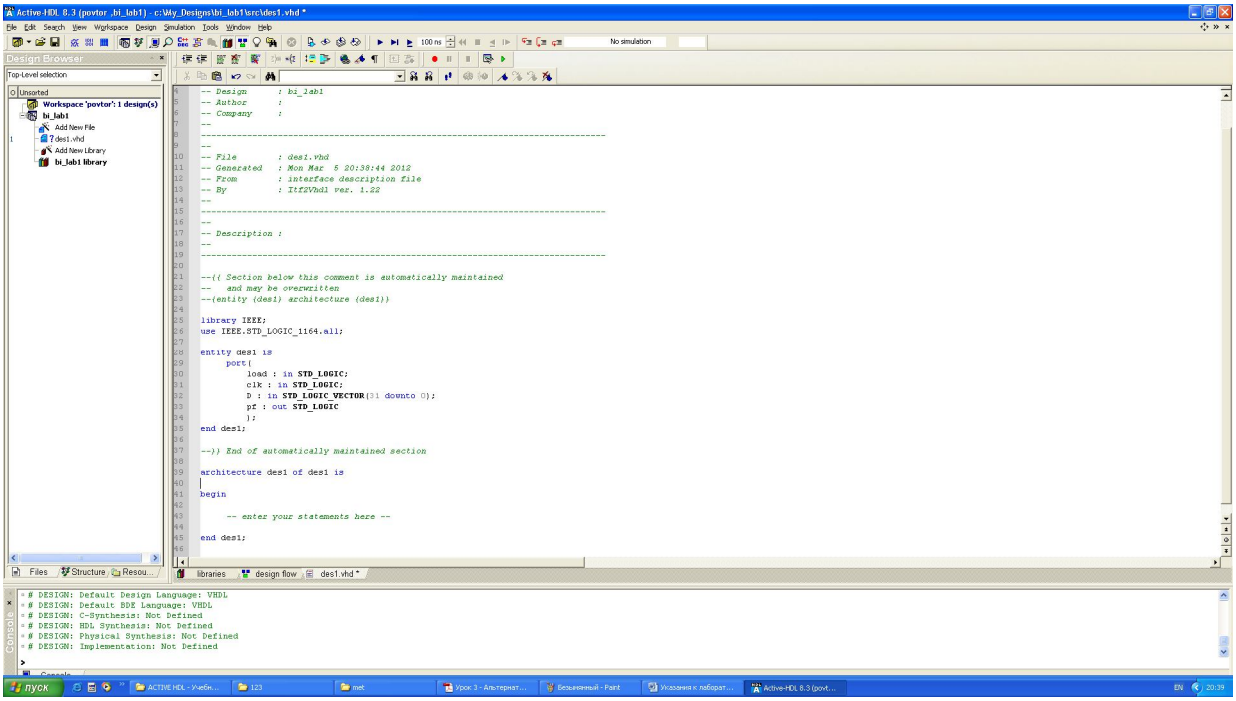

Рисунок 1.14 – итоговое окно мастера создания проекта.

Внутри парного оператора entity is…end содержится информация о структуре создаваемого блока, в данном случае имена и типы входных и выходных портов, заданных в пункте 12. Назначение выводов создаваемого компонента: **clk** – вход тактовой частоты, **load** – вход загрузки данных, **SDi** – вход шины данных, pf – выход, характеризующий состояние бита чётности.

Внутри парного оператора architecture is…end (рис.1.15) после оператора begin вставляется информация, характеризующая алгоритм работы блока в коде VHDL.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity des1 is
     port(
          load : in STD_LOGIC;
          clk : in STD_LOGIC;
          D : in STD_LOGIC_VECTOR(31 downto 0);
          pf : out STD_LOGIC
          );
end des1;

--}} End of automatically maintained section

architecture des1 of des1 is

begin

     -- enter your statements here --

end des1;
```

Рисунок 1.15 – начальный вид VHDL-кода.

13. Опишите в коде VHDL работу схемы (рис.1.16), осуществляющей вычисление бита чётности для поступающих в параллельном коде двойных слов. При переводе схемы в код используйте помощь - окно Language Assistant, появляющееся при нажатии пиктограммы  на панели инструментов, инструментарий Synthezis Templates.

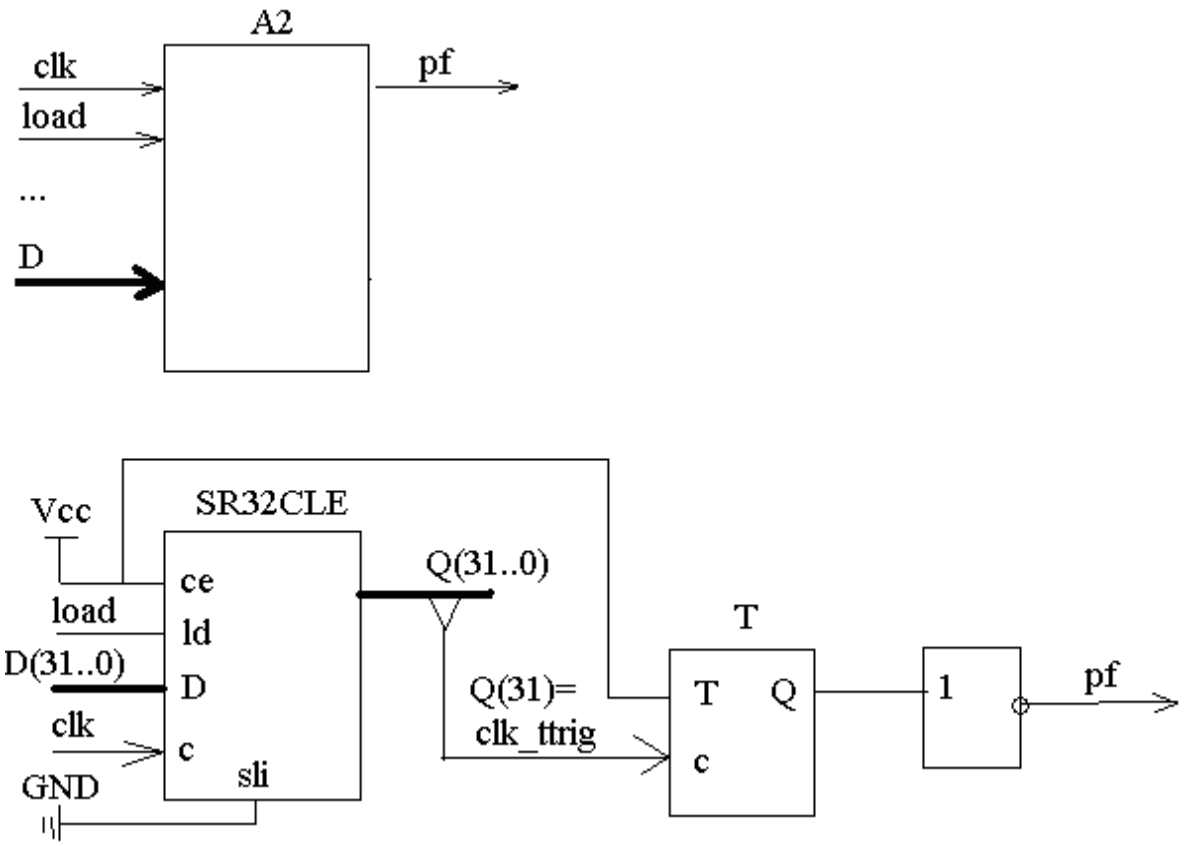

Рисунок 1.16 – схема задания.

A2 – создаваемый компонент, внутренняя структура которого включает сдвиговый регистр SR32CLE, Т-триггер T, выходной инвертор.

14. Произведите компиляцию кода нажав на кнопку Compile  . . Компиляция успешна при появлении в окне **Console** сообщения **Compile success.**

15. После прохождения этапа компиляции создайте TestBench, для чего выберите Tools->Generate TestBench. В появившемся окне **Testbench Generator Wizard** (рисунок 1.17) в поле Entity введите имя вашего модуля, с которым должен ассоциироваться тестовый файл. В поле Testbench Type выставьте флажок Single Process.

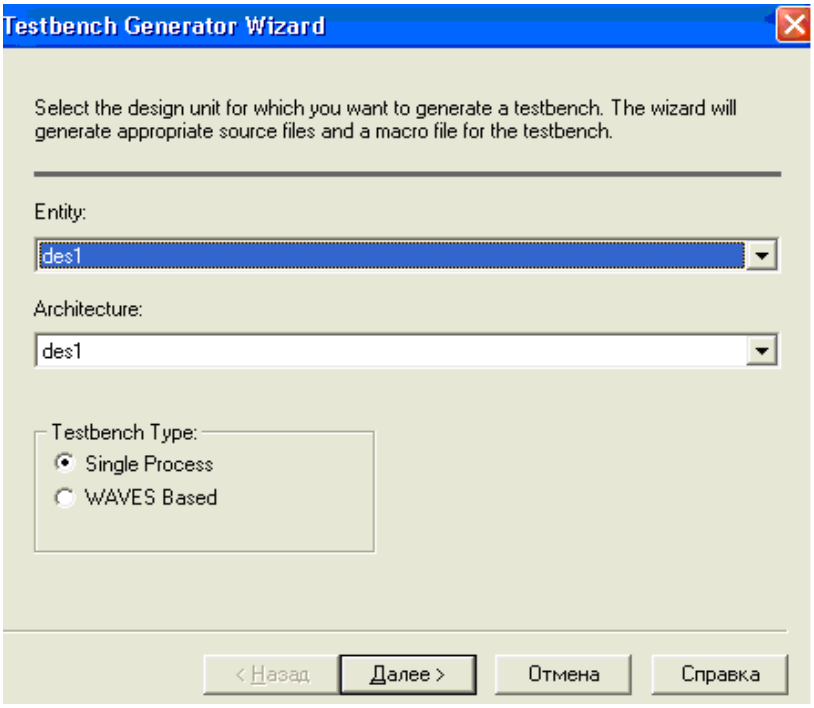

Рисунок 1.17 – выбор структурной составляющей проекта.

16. В появившемся окне (рисунок 1.18) нажмите кнопку **Далее**.

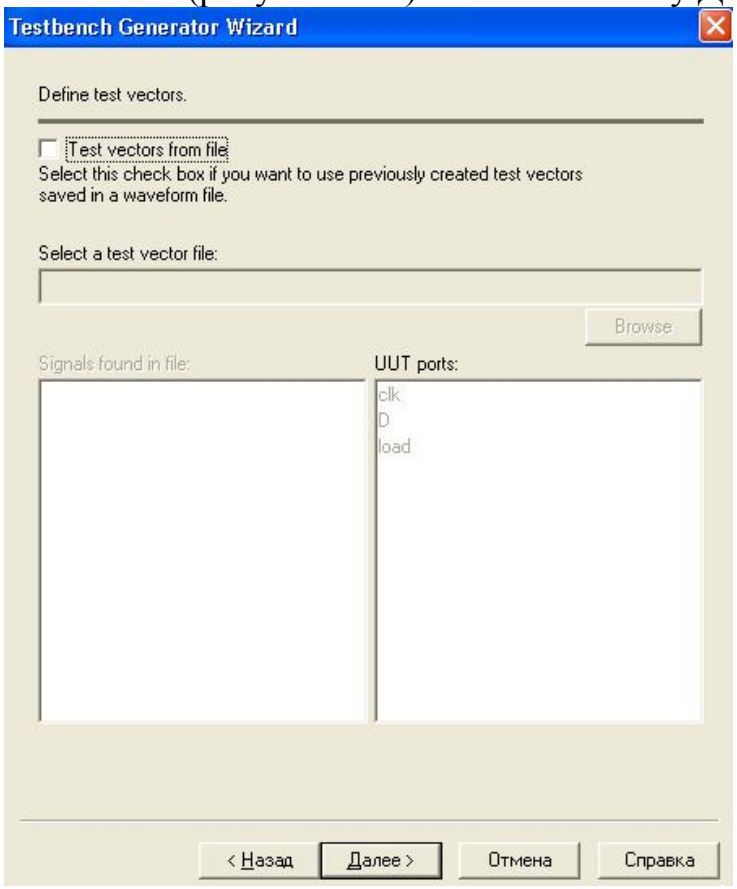

Рисунок 1.18 – формирование тестового вектора.

17. В появившемся окне (рисунок 1.19) нажмите кнопку **Далее**.

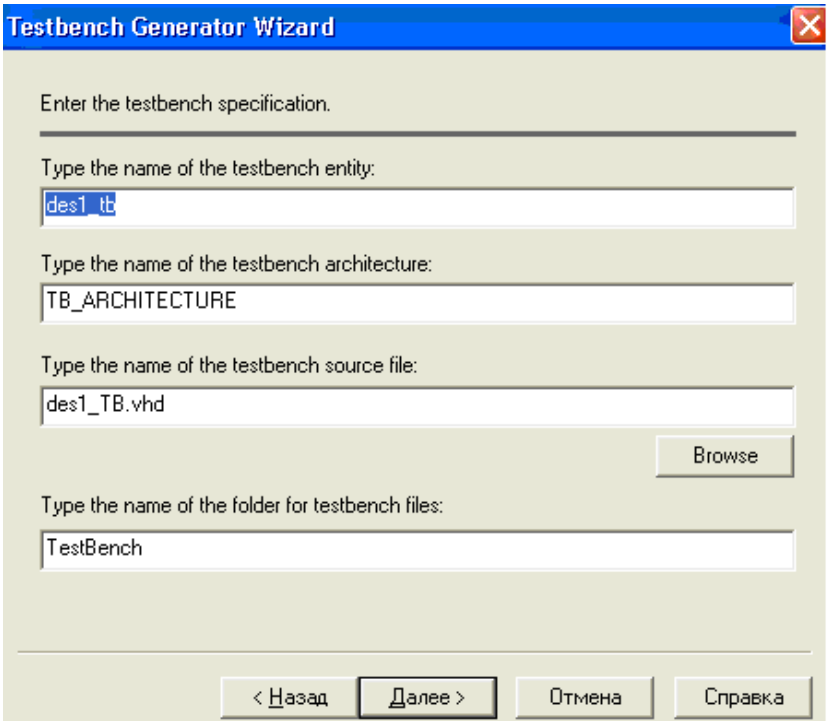

Рисунок 19 – ввод имени тестового вектора.

18. В появившемся окне (рисунок 1.20) выставьте флажок Generate. нажмите кнопку **Готово**.

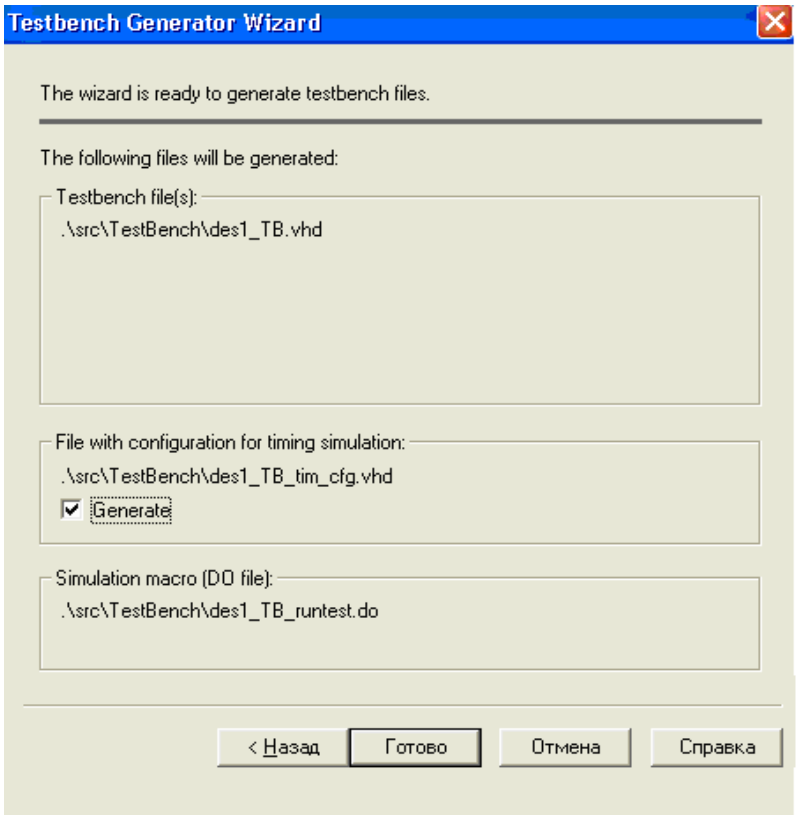

Рисунок 1.20 - окончание формирование тестового вектора.

19. В появившемся окне кода проинициализируйте следующие сигналы:

```
signal clk : STD_LOGIC:='0';
signal load : STD_LOGIC:='0';
signal D : STD_LOGIC_VECTOR(31 downto 0):="01010101010101010101010101010101";
-- Observed signals - signals mapped to the output ports of tested entity
signal pf : STD_LOGIC;
```

20. Вставьте после строки **Add your stimulus here ...** следующий код:

```
tb : PROCESS
BEGIN
        -- Wait 100 ns for global reset to finish
        wait for 100 ns;
        for i in 0 to 40 loop
                clk<=not(clk);
                if i=3 then load<='1'; D<="01010101010101010101010101010101";
                else load<='0'; D<="00000000000000000000000000000000"; end if;
                wait for 10 ns;
                clk<=not(clk);
                wait for 10 ns;
        end loop;
        -- Place stimulus here
        wait; -- will wait forever
END PROCESS;
```

21. Сохраните, откомпилируйте созданный Testbench и запустите на исполнение файл *.do в Design Brouser. Появляется стартовое окно моделирования (рисунок 1.21).

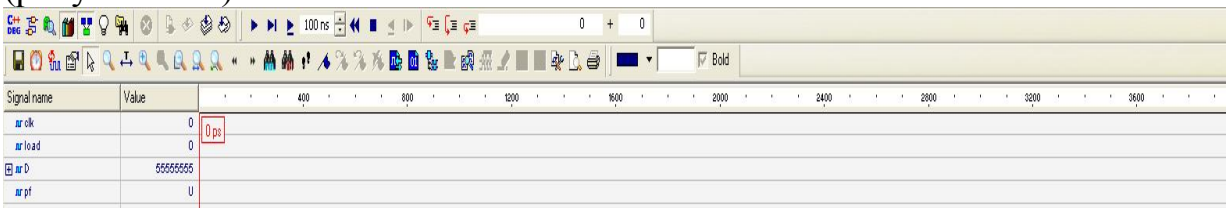

Рисунок 1.21 – начальное окно моделирования.

22. Запустите процесс моделирования на исполнение нажатием кнопки RUN  . Результатом работы схемы (рис.1.16) будет состояние выхода pf по окончании моделирования (рисунок 1.22).

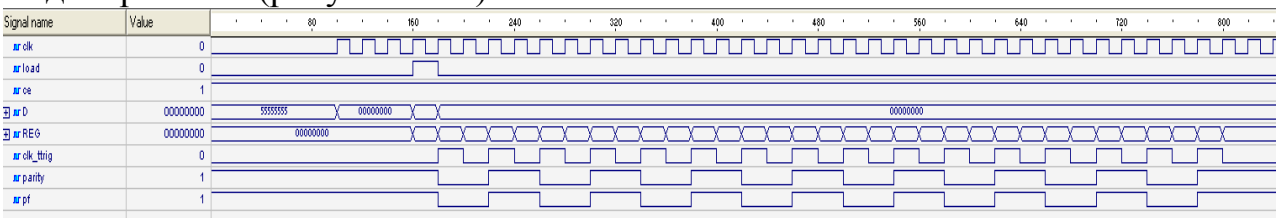

Рисунок 1.22 – временная диаграмма.

23. Попробуйте, видоизменяя тестовый файл, проверить работу схемы на других примерах двойных слов.

24. Предложите более простую схему формирования бита чётности.

Лабораторная работа №2

Формирование CRC-последовательностей.

Цель работы: развитие навыков проектирования схемных решений для вычисления контрольных характеристик информационных последовательностей, моделирования и отладки разработанных решений в САПР Active-HDL 8.3.

Задание на лабораторную работу.

Создать и провести моделирование работы структурного блока (A2 на рисунке 1), выполняющего функцию CRC-последовательности для двойного слова, поступающего в параллельном коде на вход блока A2 по шине SDi в соответствии с временной диаграммой на рисунке 2.1. Среда моделирования ActiveHDL 8.3.

Порядок выполнения работы:

1. Создайте проект как в лабораторной работе №1, задав при этом выводы, соответствующие следующему порту:

```
port(
        load : in STD_LOGIC;
        clk : in STD_LOGIC;
        SDi : in STD_LOGIC_VECTOR(31 downto 0);
        REZ: out STD_LOGIC;
        CRC : out STD_LOGIC_VECTOR(15 downto 0));
```

Назначение выводов создаваемого компонента: clk – вход тактовой частоты, load – вход загрузки данных, SDi – вход шины данных, REZ – признак вычисления результата, CRC – выходная последовательность CRC.

2. Реализуйте прямой CRC-алгоритм (или какой-либо другой), описанный в разделе “основные теоретические сведения”, на языке VHDL, создав компонент A2 (рис.1). Откомпилируйте код полученного компонента.

Варианты образующего полинома:

1. 1001000110100101.
2. 1110110000001001.
3. 1000000110000011.
4. 1000010110100001.

5. 1000000101010101.
6. 1001100100010001.
7. 1011000111011101.
8. 1100000000000101.
9. 1000000111000001.
10. 1111110101010101.

3. Создайте тестовый пример (Testbench), соответствующий рисунку 1. Произведите моделирование работы компонента.

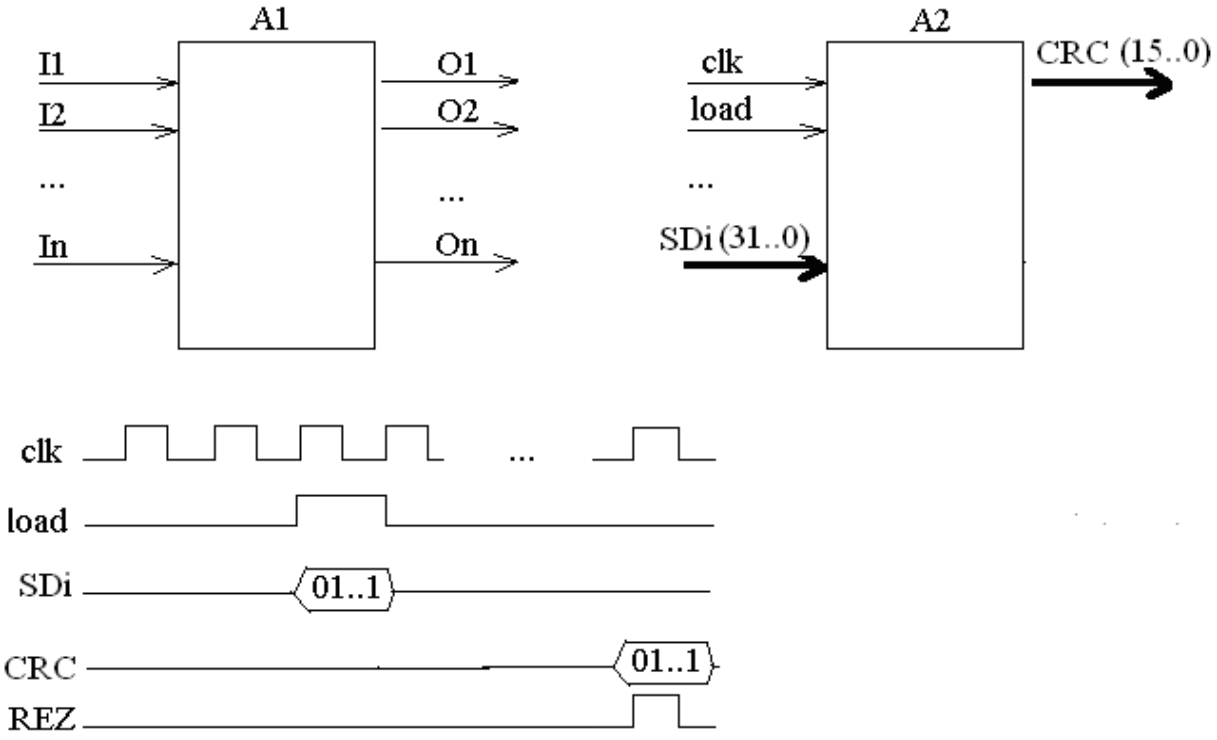

Рисунок 2.1 – структурный элемент для проектирования.

Лабораторная работа №3

Аппаратная реализация битстаффинга.

Цель работы: развитие навыков проектирования, отладки и моделирования схемных решений в САПР Active-HDL 8.3 применительно к процедуре битстаффинга.

Задание на лабораторную работу.

Создать и провести моделирование работы структурного блока (A1 на рисунке 1а), выполняющего следующие функции:

1. Преобразование параллельного кода в последовательный.

2. Поиск и исключение из кода последовательности бит, совпадающей с флагом (согласно варианту) путём добавления в найденную последовательность “разбивающего” бита – битстаффинг (рис.3.1в, 3.1г).

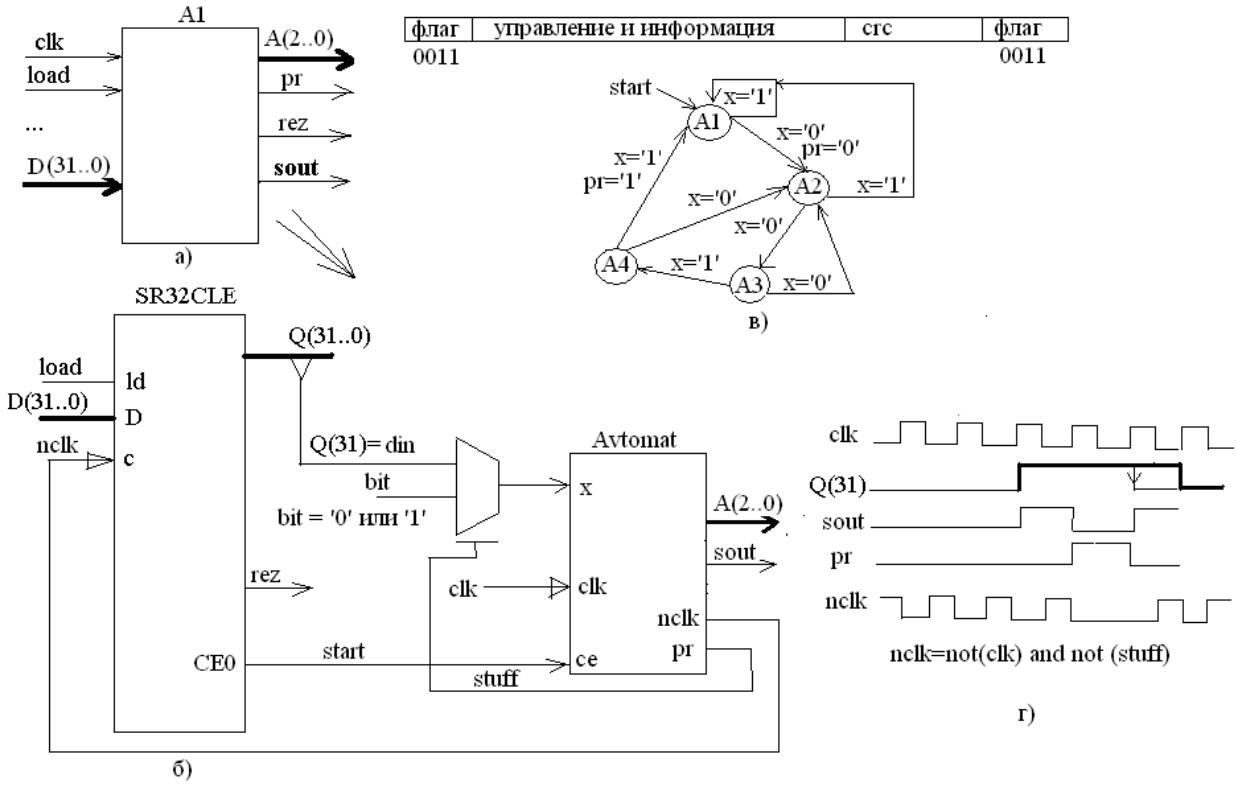

Рисунок 3.1 - структурные элементы для проектирования.

Порядок выполнения работы:

1. Постройте автомат (пример - рис.3.1в) для поиска в последовательном коде совпадающих с флагом (согласно варианту) комбинаций.

Варианты флагов:
11.00001111.
12.11110000.
13.10000001.
14.01111110.
15.01110110

2. Создайте проект как в лабораторной работе №1, задав при этом выводы, соответствующие следующему порту:

```
port(
        clk : in STD_LOGIC;
        load : in STD_LOGIC;
        D : in STD_LOGIC_VECTOR(31 downto 0);
        sout : out STD_LOGIC;
        pr : out STD_LOGIC;
        rez : out STD_LOGIC;
        A : out STD_LOGIC_VECTOR(2 downto 0) )
```

Назначение выводов создаваемого компонента:
clk – вход тактовой частоты,
load – вход загрузки данных,
D[31..0] – вход параллельной шины данных,
A[2..0] – идентификатор состояний разработанного автомата,
pr – признак наличия в коде искомой последовательности,
sout – выход данных в последовательном коде,
rez – признак выполнения задачи.

3. Создайте на основе сдвигового регистра компонент SR32CLE (рис.3.1б), осуществляющий преобразование из параллельного кода в последовательный, запуск автомата и выдачу признака выполнения задания rez.

4. Реализуйте в виде процесса автомат (компонент Avtomat на рисунке 3.1в), соответствующий спроектированному в пункте 1 автомату.

5. Создайте тестовый пример (Testbench), подав на входы воздействия как на рисунке 3.2, и произведите моделирование работы компонента A1.

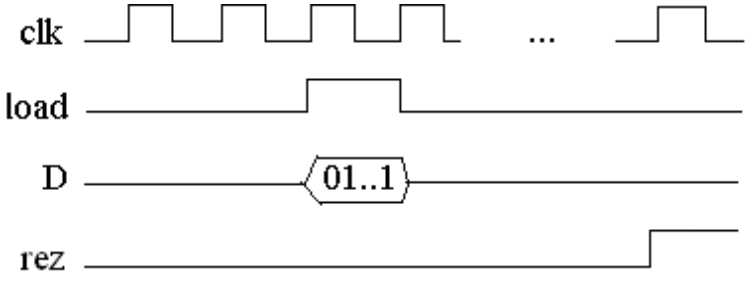

Рисунок 3.2 – временная схема воздействий для моделирования компонента.

Лабораторная работа №4

Аппаратная реализация процедуры, обратной битстаффингу.

Цель работы: развитие навыков проектирования, отладки и моделирования схемных решений в САПР Active-HDL 8.3 применительно к процедуре восстановления информационных последовательностей.

Задание на лабораторную работу.

Создать и провести моделирование работы структурного блока (A1 на рисунке 4.1а), выполняющего следующие функции:

1. Поиск и исключение из кода последовательности бита, разбивающего флаг в лабораторной работе №3 флагом (согласно варианту).
2. Преобразование последовательного кода в параллельный.

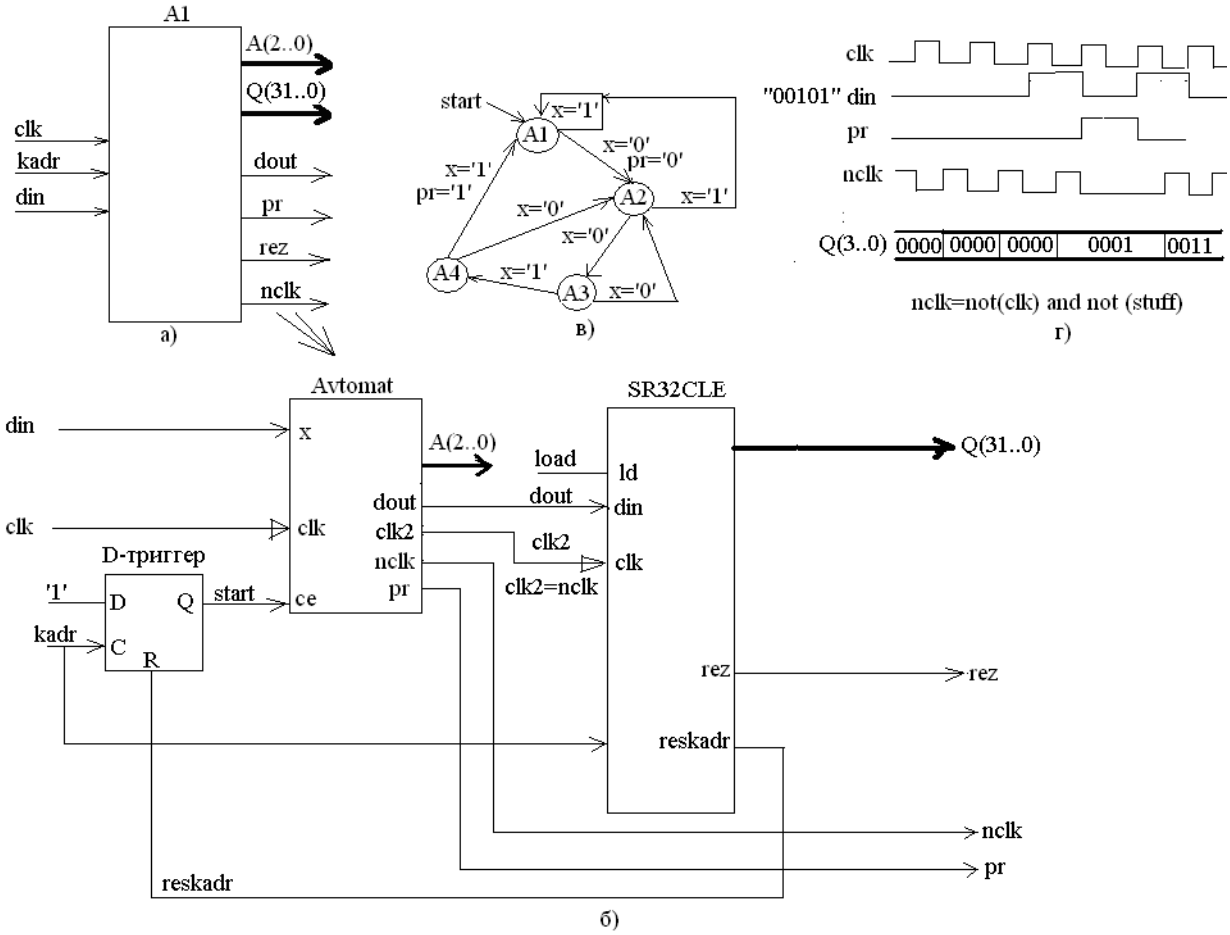

Рисунок 4.1 - структурные элементы для проектирования.

Порядок выполнения работы:

1. Постройте автомат (пример - рис.4.1г) для поиска в последовательном коде совпадающих с флагом (согласно варианту) комбинаций.
   Варианты флагов:
   16.00001111.
   17.11110000.
   18.10000001.
   19.01111110.

2. Создайте компонент A1 на рисунке 4.1. Для этого создайте проект как в лабораторной работе №1, задав при этом выводы, соответствующие следующему порту:

```
port(
        clk : in STD_LOGIC;
        din : in STD_LOGIC;
        kadr : in STD_LOGIC;
        prd : out STD_LOGIC;
        dout : out STD_LOGIC;
```

```
        rez : out STD_LOGIC;
        nclk : out STD_LOGIC;
        A: out STD_LOGIC_VECTOR(2 downto 0);
        Q : out STD_LOGIC_VECTOR(31 downto 0)
    )
```

Назначение выводов создаваемого компонента:
clk – вход тактовой частоты,
din – вход параллельной шины данных,
kadr - признак кадра, стробируюший работу автомата.
sout - выход данных в последовательном коде,
A[2..0] – идентификатор состояний разработанного автомата,
pr – признак наличия в коде искомой последовательности,
Q[31..0] – выход данных в параллельном коде,
rez – признак выполнения задачи.

3. Реализуйте в виде процесса автомат (компонент Avtomat на рисунке 4.1б), соответствующий спроектированному в пункте 1 автомату.

4. Создайте на основе сдвигового регистра компонент SR32CLE (рис.4.1б), осуществляющий преобразование из последовательного кода в параллельный и выдачу признака выполнения задания rez.

5. Создайте тестовый пример (Testbench), соответствующий рисунку 4.2. Произведите моделирование работы компонента A1.

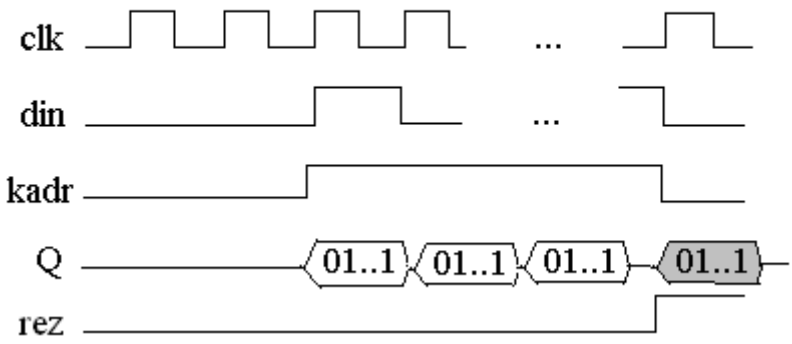

Рисунок 4.2 – временная схема воздействий для моделирования компонента.

3 ОФОРМЛЕНИЕ ОТЧЕТА

После выполнения каждой из лабораторных работ студентом оформляется отчет и представляется преподавателю для проверки с последующей защитой (выполнение отчета и защита работы проводится каждым студентом индивидуально).

Работа оформляется в последовательности, приведенной в методических указаниях.

На первой странице пишется заглавие, указывается цель и объем работы в часах, Ф.И.О. студента, группа, дата выполнения.

Текст работы оформляется на ПЭВМ шрифтом Times New Roman с использованием средств текстового процессора и выводится на принтер на листах формата А4 (210 * 297 мм) с соблюдением ГОСТ 2.105-95, ГОСТ 8.417-2002 и ГОСТ 7.1-2003.

В отчете по проделанной работе должны быть включены следующие структурные элементы:

а) титульный лист;

б) цель работы;

в) основная часть, содержащая постановку задачи и полученные результаты, а также отражающая процесс выполнения работы;

г) выводы.

Перенос слов на титульном листе и в заголовках текста не разрешается. Точка в конце заголовка не ставится.

Защита лабораторных работ осуществляется по результатам выполненного задания, в процессе защиты выполняется дополнительная проверка (с использованием контрольных вопросов) усвоения студентом материала.

ЛИТЕРАТУРА

1. Бортовые информационные системы: курс лекций / А.А. Кучерявый – Ульяновск: Изд-во УГТУ, 2004. – 160 с.