Лабораторная работа 1
По курсу: «Компьютерная схемотехника»

Проектирование арифметико-логического устройства

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

2. **Теоретические сведения**

Устройство LSM предназначено для выполнения как арифметических действий (сложение, вычитание) так и логических действий (побитное И, ИЛИ, НЕ, Исключающее ИЛИ) над данными, представленными параллельными *n*-разрядными кодами с фиксированной запятой, например, *A* и *B*. Чаще всего эти данные представлены в дополнительном коде. В качестве особого операнда выступает бит C_0 переноса в младший разряд. Кроме *n*-разрядного результата *Y*, результатами LSM часто выступают такие признаки результата, как перенос из старшего разряда *C*n, признак переполнения *V*, признак нулевого результата *Z* и бит знака *S*.

Тип операции LSM задается управляющим кодом *F*, кодировка которого выбирается в каждом конкретном случае особо.

3. **Элементная база**

При выполнении лабораторной работы предлагается реализовать LSM на базе современной сложной ПЛМ (CPLD) или программируемой логической интегральной схемы (ПЛИС или FPGA). В первом случае в качестве базового элемента используется элемент ПЛМ (PLM), а во втором – программируемая логическая таблица (LUT).

Один элемент PLM может выполнить любую логическую функцию, которая представляется дизъюнкцией от *M* конституент единицы (термов) или ее инверсией, причем у каждой конституенты может быть не более *N* булевских переменных или их инверсий, а число различных входных булевских переменных не может быть больше *K*. Для современных CPLD, таких как Altera MAX7000, Xilinx X9500 параметры максимальных размеров PLM удовлетворяют неравенствам: $M \le 5$, $N \le 52$, $K \le 52$. Для увеличения M используются PLM –логические расширители, что эквивалентно последовательному включению PLM.

Например, допустимо такое задание логической функции (на языке VHDL) :

```
Y<=not((not A and B) or (B and C and not D) or
 (A and C and not E and F) or (B and C and F and not G and L);
```

для которой $M=4$, $N=5$, $K=9$. Здесь A, B, C, D, E, F, G, L – входные булевские или битовые переменные, а Y – выходной сигнал-результат. Ключевые слова `and, or` и `not` обозначают операции И, ИЛИ и НЕ, соответственно. Такой оператор присваивания сигналу соответствует функциональной схеме на рис.1.

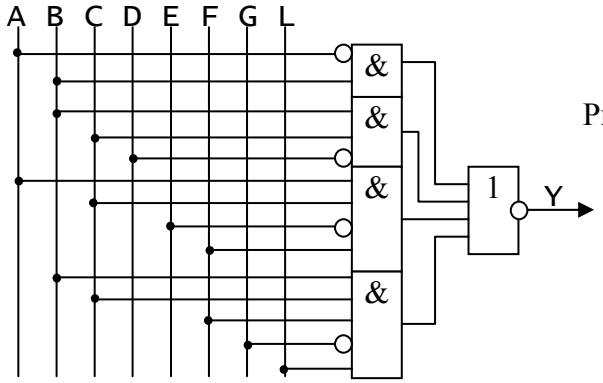
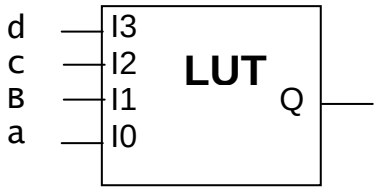


Рис.1. Функциональная логическая схема, соответствующая оператору VHDL.

Программируемая логическая таблица (LUT – lookup table) представляет собой однобитное постоянное запоминающее устройство на 2^K ячеек. Причем в ячейке по адресу *i* хранится 1, если в совершенной дизъюнктивной нормальной форме (СДНФ) логической функции присутствует конституента единицы от всех *K* входных адресных битов, соответствующая этому адресу . При этом слово адреса *i* формируется таким образом, что если в конституенте стоит переменная с инверсией, то соответствующий бит адреса – нулевой, а иначе – он единичный. Например, следующая СДНФ

$$Y = \bar{d}c\bar{b}a \vee \bar{d}cba \vee dc\bar{b}a ;$$

кодируется в LUT как единица, записанная по адресу 0101_2 =5, 0111_2 =7 и 1101_2 =13. На рис.2. показана LUT, в которой закодирована эта функция. В современных ПЛИС применяются 3, 4 и 5 – входовые LUT.




```
Q:=(not d and c and not b and a) or
   (not d and c and b and a) or
    (d and c and not b and a) ;
```

d	c	b	a	Q
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Рис.2. Логическая таблица для исполнения заданной функции

3. **Библиотека функций и элементов для описания LSM**

Лабораторная работа проводится с применением языка VHDL и симулятора этого языка – системы Active HDL. Для удобства проведения лабораторных работ и в частности, для моделирования LSM, была разработана библиотека функций и элементов, описанная на VHDL. Библиотека хранится в файле CNetlist_Lib.VHD.

В библиотеке хранится пакет функций CNetlist и модели LUT, LSM , а также модель генератора случайных чисел.

Пакет в VHDL представляет собой набор констант, функций, процедур, которые могут быть использованы в различных VHDL – программах. Пакет состоит из декларативной части, в которой приводятся объявления констант и интерфейсы процедур и функций, и описательной части, в которой описывается исполнение процедур и функций, объявленных в декларативной части. Для правильного пользования пакетом достаточно знать его декларативную часть, в которой описано, как процедуры и функции включать в VHDL - программы. Декларативная часть пакета функций выглядит следующим образом.

```
Package CNetlist is
 --преобразование вектора бит - числа в доп.коде в целое
 function BIT_TO_INT(b:bit_vector) return integer;
 --преобразование бита в целое
 function BIT_TO_INT(b:bit) return integer;
 --преобразование целого в вектор бит - число в доп.коде
 function INT_TO_BIT(i,l:integer) return bit_vector;
 --преобразование целого 0|1 в бит
 function INT_TO_BIT(i:integer) return bit;
 --реализация логической табличной  функции
 --   e,d,c,b,a - образуют адрес в таблице, e- старший бит,
 -- mask-содержимое таблицы, левый бит- по старшему адресу
function LOG_TAB(e,d,c,b,a:BIT:='0';mask:bit_vector) return bit;
 end CNetlist;
```

Этот пакет содержит функции `BIT_TO_INT(b)` преобразования битового представления числа `b` в целочисленное, функции `INT_TO_BIT(i,l)` преобразования целого числа `i` в число в дополнительном коде, содержащем заданное количество бит `l`. Назначение функций и типы входных переменных и результатов понятны из комментариев к ним. Функция `LOG_TAB(e,d,c,b,a,mask)` возвращает значение логической функции от входов `e,d,c,b,a`, реализованную в виде таблицы, содержимое которой задается вектором битов `mask`. Например,

`BIT_TO_INT("01111")` - возвращает 15,
`BIT_TO_INT("10001")` - возвращает -15,
`INT_TO_BIT(5,4)` - возвращает – "0101",
`INT_TO_BIT(-5,4)` - возвращает – "1001",
`LOG_TAB(open,'0','0','0','1',"0111111111111111")` – возвращает '1', т.е. функцию ИЛИ от четырех входов. Здесь вектор битов определяет содержимое LUT, причем левый бит соответствует ячейке с нулевым адресом, ключевое слово **open** означает, что данный вход функции не используется и принимается равным 0.

Пакеты и объекты VHDL - проекта группируются в одну библиотеку после своей компиляции. Объект проекта – это основная программная единица в VHDL, в которой описано, как включать объект в некоторый проект или схему, т.е. его интерфейс (объявление объекта) и каково его поведение в зависимости от входных сигналов и состояний объекта (архитектура объекта).

Приведенный в библиотеке вариант модели элемента PLM имеет следующее описание интерфейса (объявление объекта).

```
entity PLM_4 is
 generic(
 td:time:=1 ns);                -- задержка
      A : in BIT;               -- входные переменные
      B : in BIT;
      C : in BIT;
      D : in BIT;
      Y : out BIT); -- результат
end PLM_4;
```

Объявление объекта PLM описывает, как этот объект "выглядит снаружи", т.е. указывает, каким образом его включать в электрическую схему, если этот объект вставлять в нее как компонент.

В нем фраза **entity** объявляет имя объекта проекта - PLM. В фразе **generic** перечисляются настроечные переменные объекта PLM, причем n обозначает длину входного вектора, т.е. число входных битовых переменных, а td - задержку PLM. В этой же фразе указаны начальные значения настроечных переменных, которым могут быть присвоены новые значения при связывании этих переменных, когда объект PLM будет вставляться как компонент в объект более высокого уровня.

В фразе **port** перечисляются сигналы, которые являются входным A и выходным Y сигналами объекта PLM. При этом A – это вектор битов длиной n, задаваемой как настроечная переменная. Его разряды нумеруются справа - налево , начиная 0 и кончая n-1. Настроечная переменная n означает, что при вставке компонента PLM , вектор входных логических переменных можно задать конкретной длины, задаваемой целым положительным числом. Ключевые слова **in** и **out** указывают на направление передачи информации: в объект или из объекта.

Алгоритм функционирования объекта PLM описан в следующем примере его архитектуры.

```
architecture STENCIL of PLM_4 is
begin
 Y<=not(          -- инверсия результата, необязательно
    (not D and not C and not B) -- первый терм
    or (not B and A)             -- второй терм...
    or (B and not A)
    or ( C and A)
    or D
    )          after td; -- задержка
end STENCIL;
```

Архитектура STENCIL объекта PLM, обозначаемая как PLM(STENCIL), описывает один из многих вариантов реализации объекта. В данном случае объект описан с помощью параллельного оператора присваивания сигналу Y логического выражения от входных битов D,C,B,A, причем результат выражения задерживается на td наносекунд. Как видим, логическое выражение имеет 5 термов, объединенных функцией ИЛИ и инверсию результата, т.е. такое выражение отображается в один элемент PLM. Его графическое представление показано на рис.3.

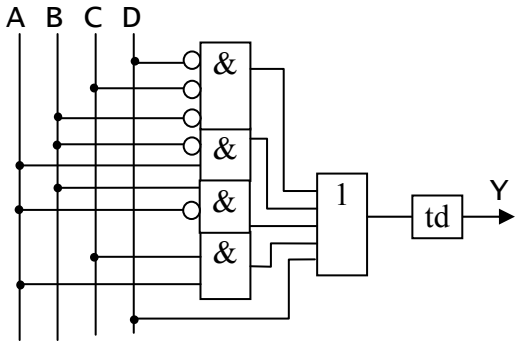


Рис. 3. Функциональная логическая схема элемента PLM

В VHDL каждый объект проекта описывается парой: объявление объекта – ENTITY и описание объекта – ARCHITECTURE, или просто – архитектура. Говорят, что если архитектура описана параллельными операторами присваивания, такими как приведенный выше, то она описана стилем потоков данных.

В том же файле библиотеки описаны объекты – модели четырех- и пятивходовых элементов LUT. Их объявления приведены ниже.

```
use Cnetlist.all;
entity LUT4 is
 generic(mask:bit_vector(15 downto 0):=X"ffff";
 td:time:=1 ns);
 port(a : in BIT;
      b : in BIT;
      c : in BIT;
      d : in BIT;
      Y : out BIT);
end LUT4;

use Cnetlist.all;
entity LUT5 is
 generic(mask:bit_vector(31 downto 0):=X"ffffffff";
 td:time:=1 ns);
 port(a : in BIT;
      b : in BIT;
      c : in BIT;
      d : in BIT;
      e : in BIT;
      Y : out BIT);
end LUT5;
```

Здесь описание **use** указывает компилятору, что для компиляции объекта необходимо использовать пакет Cnetlist, а ключевое слово **all** – что в этом пакете могут быть использованы все его составляющие элементы. Файл Cnetlist_Lib.VHD с пакетом должен находиться в рабочем каталоге и скомпилирован в рабочую библиотеку, которая неявным образом именуется как work.

В разделе настроечных констант **generic** указывается содержимое логической таблицы в виде вектора бит mask с диапазоном :(15 downto 0). Вектор бит может представляться в двоичном виде или шестнадцатиричном виде. При этом самый правый бит в векторе (с номером 0) заносится в младшую ячейку таблицы. Например, функция ИЛИ от четырех переменных кодируется следующими литералами: "0111111111111111" или X"7FFF".

Генератор случайных чисел используется в лабораторных работах для подачи на вход спроектированных LSM тестовых последовательностей. Он имеет следующее объявление объекта:

```
Library IEEE;
Use IEEE.MATH_REAL.ALL;
 use Cnetlist.all;
entity RANDOM_GEN is
 generic(n:positive:=8;     -- разрядность выходного слова
         tp:time:=100 ns;   -- период следования
       SEED:positive:=12345 -- начальное состояние
 );
 port(CLK: out BIT;
        Y : out BIT_vector(n-1 downto 0));
end RANDOM_GEN;
```

Для описания этого объекта используется библиотека IEEE, на что указывает описание **Library**. Из этой библиотеки используется пакет MATH_REAL, в котором хранятся константы и функции, необходимые для математических вычислений с плавающей запятой. В частности, в генераторе используется процедура:

```
procedure UNIFORM(variable SEED1,SEED2:inout POSITIVE;
                  variable X:out real);
```

при обращении к которой целые переменные SEED1,SEED2 случайным образом изменяют свое значение и от них вычисляется случайное число X с плавающей запятой в диапазоне от 0,0 до 1,0.

В объявлениях настроечных констант указываются n – разрядность выходного вектора бит Y, представляющего целое случайное число, tp - период следования (изменения) выходных данных. Каждое выдаваемое данное стробируется выходным сигналом CLK. Для генератора необходимо указывать начальное состояние SEED, для того, чтобы несколько генераторов не выдавали одинаковые последовательности данных.

4. **Примеры описания LSM**

Рассмотрим пример проектирования n=4 –разрядного LSM, который при F=00 выполняет операцию A+B+C0, при F=01 – операцию A-B-C0, при F=10 – операцию поразрядного И над A и B и при F=11 – операцию поразрядного ИЛИ, где C0 – перенос в младший разряд. При этом кроме результата Y, выдается бит Z – признак нулевого результата и бит C3 – перенос из старшего разряда.

Объявление объекта для такого LSM выглядит следующим образом.

```
use  Cnetlist.all;
entity LSM is
 port(F : in BIT_VECTOR(1 downto 0);-- функция
      A : in BIT_VECTOR(3 downto 0);-- первый операнд
      B : in BIT_VECTOR(3 downto 0);-- второй операнд
      C0: in BIT;                   -- вход переноса
      Y : out BIT_VECTOR(3 downto 0);-- результат
      C3: out BIT;                  -- выход переноса
      Z : out BIT       -- признак нулевого результата
      );
end LSM;
```

4.1. Поведенческая модель LSM.

В своем поведенческом описании объект представляется в виде алгоритма функционирования, не привязанном к конкретной элементной базе. Но, в отличие от алгоритма, заданного в виде программы на обычном языке высокого уровня, здесь задается четкая последовательность операций, выполняемых во времени. При этом операции выполняются не последовательно друг за другом, а последовательно- параллельно: последовательные операторы выполняются последовательно, а параллельные – параллельно. В данной лабораторной работе устройство LSM описывается параллельными операторами стилем потоков данных.

Поведенческую модель LSM можно описать в виде следующего описания архитектуры.

```
architecture BEH of LSM is
 signal ai,bi,ci,yi:integer;
 signal ybi:BIT_VECTOR(4 downto 0);
begin
 -- представляем вх. данные целыми
 ai<= BIT_TO_INT(A);
 bi<= BIT_TO_INT(B);
 ci<= BIT_TO_INT(C0);
-- Сумматор - вычитатель
ADDER:yi<= ai+bi+ci when F(0)='0' else
           ai-bi-ci;
```

```
-- Мультиплексор результата
MUX:with F select
 ybi<= INT_TO_BIT(yi,5) when "00"|"01",--сумматор-вычитатель
        '0'&(A and B) when  "10",    -- Схема И
        '0'&(A or  B) when others;    --Схема ИЛИ
 C7<= ybi(4);                  --          выходной перенос
 Z<='1' when ybi(3 downto 0)="0000" else '0';--призн.нуля
 Y<= ybi(3 downto 0);          --           Результат
end BEH;
```

В части объявлений в архитектуре объявлены сигналы, которые используются как промежуточные сигналы в вычислениях. В языке VHDL операции с переменными могут выполняться только в последовательных операторах, которые стоят внутри такого параллельного оператора, как process или в теле процедуры или функции. В отличие от переменной, сигнал переносит значение от одного параллельного оператора к другому и вместе с ним – синхронизирующее воздействие. Сигнал можно запомнить в его развитии и затем воспроизвести в виде графика. Порты в объявлении объекта выступают в роли сигналов.

После части объявлений в архитектуре, следует описательная часть между ключевыми словами **begin** и **end**. В этой части размещают параллельные операторы. Так как все параллельные операторы выполняются одновременно, то их порядок написания не имеет значения. Но для удобства чтения, параллельные операторы выстраивают в порядке прохождения данных через них, т.е. в направлении потоков данных и последними операторами ставят операторы присваивания выходным портам. Поэтому стиль, когда все описание - из параллельных операторов присваивания, называют стилем потоков данных.

Оператор параллельного присваивания обозначается символами: <=.

Вначале выполняется преобразование типов сигналов – сигналов векторов бит `A, B, C0` в сигналы `ai,bi,ci` целого типа с помощью функций преобразования из пакета `Cnetlist`. В операторе условного параллельного присваивания, обозначенном меткой `ADDER`, описывается сумматор-вычитатель, который в зависимости от условия `F(0)`, сложение или вычитание целых чисел. Следует отметить, что в VHDL все параллельные операторы можно отметить меткой, имя которой упрощает чтение и отладку программы.

В операторе селективного параллельного присваивания, обозначенном меткой `MUX`, описывается мультиплексор, который по коду вектора `F` выбирает или выход сумматора –вычитателя, или функцию И, или функцию ИЛИ от разрядов входных операндов. При этом результат сложения- вычитания преобразуется в 5- разрядный вектор с учетом разряда переполнения, а результаты логических операций дополняются нулем в старших разрядах. Дополнение нулем выполняется с помощью операции конкатенации бит: `'0'&(A or B)`. Такое дополнение необходимо, потому что при присваивании вектору бит, разрядность вектора `ybi,` стоящего слева от оператора `<=` должна быть равна разрядности результата выражения справа от этого оператора. Обобщая это правило, следует сказать, что в VHDL во всех операторах присваивания сигналов и переменных, а также фразах связывания портов, агрегатов и настроечных переменных, операнды слева и справа от знака присваивания или связывания должны совпадать как по типу, так и по разрядности.

Результат `Y` формируется как 4 младших разряда сигнала `ybi`, а результат `C` – как старший разряд этого сигнала. Признак нулевого результата `Z` формируется как результат операции сравнения с нулем четырех младших разрядов сигнала `ybi`.

Поведенческое описание архитектуры используется для описания моделей готовых устройств или модели проектируемого устройства на этапе алгоритмического или структурного проектирования. Затем оно переписывается с целью удовлетворения требований компилятора-синтезатора, т.е. оно описывается стилем для синтеза. Такое описание, будучи скомпилированным синтезатором, дает оптимизированную логическую схему, описанную на уровне вентилей в заданном элементном базисе.

Следует отметить, что приведенный пример описания архитектуры, описан синтезируемым стилем. Но так как в нем не использованы стандартные функции преобразования типов, как, например, из пакета IEEE.Numeric_bit, то результат компиляции будет иметь слишком высокие аппаратные затраты.

4.2. Структурная модель LSM на базе LUT.

В данном примере проектируется модель LSM, описанная структурным стилем и построенная на основе элементов LUT. Такой проект может быть взаимно однозначно отображен в прошивку ПЛИС, т.е. каждый из описанных компонентов LUT будет реализован в соответствующем конкретном элементе LUT микросхемы.

LSM имеет *n* ступеней, соответствующих его разрядам. Каждую ступень можно представить объединением трех LUT, первая из них LNI , выполняет разрядное сложение или логическую операцию над входными данными *Ai*,*Bi*, вторая – LNO – прибавляет к результату *Xi* перенос *Ci* с предыдущего разряда и выдает результирующий разряд *Yi*. Третья- LNC –прибавляет к сумме или разности *Ai* и*Bi* перенос *Ci* и вычисляет перенос в следующий разряд C_{i+1}. Структура одной ступени показана на рис.4.

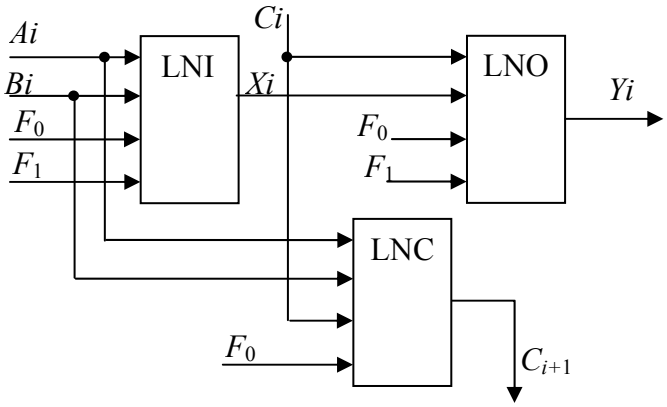


Рис.4. Структура одной ступени LSM

Указанные LUT задаются следующими таблицами истинности.

Таблица 1. Таблица истинности LNI

Адрес	Действие	F1,F0	Bi	Ai	Xi
0	A+B	00	0	0	0
1			0	1	1
2			1	0	1
3			1	1	0
4	A-B	01	0	0	1
5			0	1	0
6			1	0	0
7			1	1	1
8	A&B	10	0	0	0
9			0	1	0
10			1	0	0
11			1	1	1
12	A∨B	11	0	0	0
13			0	1	1
14			1	0	1
15			1	1	1

Таблица 2. Таблица истинности LNO

Адрес	Действие	F1,F0	Xi	Ci	Yi
0	A+B	00	0	0	0
1			0	1	1
2			1	0	1
3			1	1	0
4	A-B	01	0	0	1
5			0	1	0
6			1	0	0
7			1	1	1
8	A&B	10	0	0	0
9			0	1	0
10			1	0	1
11			1	1	1
12	A∨B	11	0	0	0
13			0	1	0
14			1	0	1
15			1	1	1

Таблица 3. Таблица истинности LNC

Адрес	Действие	F0	Ci	Bi	Ai	C_{i+1}
0	A+B	0	0	0	0	0
1			0	0	1	0
2			0	1	0	0
3			0	1	1	1
4			1	0	0	0
5			1	0	1	1
6			1	1	0	1
7			1	1	1	1
8	A-B	1	0	0	0	0
9			0	0	1	1
10			0	1	0	0
11			0	1	1	0
12			1	0	0	1
13			1	0	1	1
14			1	1	0	0
15			1	1	1	1

По правой колонке таблиц 1,2,3 можно составить содержимое соответствующих логических таблиц, которое будет равно X"E896" , X"CC96" и X"B2E8", соответственно.

Теперь можно описать одну ступень LSM следующим образом.

```
LNI:LUT4 generic map(mask=>X"6917")
         port map(a=>A(i),b=>B(i),c=> F(0),d =>F(1),
         Y =>X(i));
LNO:LUT4 generic map(mask=>X"6933")
         port map(a=>c(i),b=>X(i),c=> F(0),d =>F(1),
         Y =>yi(i));
LNC:LUT4 generic map(mask=>X"174D")
         port map(a=>A(i),b=>B(i),c=> c(i),d =>F(0),
         Y =>c(i+1));
```

Здесь применяется параллельный оператор вставки компонента. При этом в фразе **generic map** выполняется связывание настроечных переменных, а в фразе **port map** –

связывание портов компонента и сигналов. В данном примере применяется поименное связывание, когда явно указывается название порта или переменной, которые связываются с необходимыми сигналами или константами. Возможно также позиционное связывание, при котором имена портов не приводятся, но при этом возрастает вероятность совершить ошибку и затрудняется чтение программы.

Сигнал `yi` дублирует выходной сигнал - порт `Y`, так как он также используется как входной сигнал для функции определения нулевого результата и поэтому не может непосредственно подключаться к порту в режиме `out`.

Для описания всего LSM можно продублировать эти операторы *n*=4 раза, скорректировав индексы сигналов соответствующим образом. Но гораздо эффективнее это выполнить с использованием оператора **generate** – оператора размножения параллельных операторов.

Сигнал нулевого результата `Z` имеет единичное значение только на одном из 16 возможных наборов четырех битов `yi` и поэтому он может вырабатываться одной схемой LUT с таблицей X"8000".

Результирующая архитектура выглядит следующим образом.

```
architecture STR_LUT of LSM is
   signal c,x,yi: BIT_VECTOR(4 downto 0);
   component LUT4 is
     generic(mask:bit_vector(15 downto 0):=X"ffff";
             td:time:=1 ns);
     port(a : in BIT;
          b : in BIT;
          c : in BIT;
          d : in BIT;
          Y : out BIT);
   end component;
begin
        c(0)<=C0;
--  Схема арифметико-логического устройства
LSM_STR:for i in 0 to 3 generate
 LNI:LUT4 generic map(mask=>X"6917")
          port map(a=>A(i),b=>B(i),c=> F(0),d =>F(1),
           Y =>X(i));
 LNO:LUT4 generic map(mask=>X"6933")
          port map(a=>C(i),b=>X(i),c=> F(0),d =>F(1),
           Y =>yi(i));
 LNC:LUT4 generic map(mask=>X"174D")
          port map(a=>A(i),b=>B(i),c=> c(i),d =>F(0),
           Y =>c(i+1));
 end generate;
-- Определение нулевого результата
 UZ:LUT4 generic map(mask=>X"8000")
          port map(a=>yi(3),b=>yi(2),c=> yi(1),d =>yi(0),
           Y =>Z);
 Y<=yi(3 downto 0);
 C3<=c(4); --выход переноса
end STR_LUT;
```

При компиляции оператора **generate** компилятор дублирует тело оператора столько раз, сколько указано в диапазоне **for i in 0 to 3,** который пробегает индексная переменная. Эта переменная также используется как индекс векторов, биты которых привязываются к портам вставляемых компонентов. Следует отметить, что в области объявлений (до ключевого слова **begin**) необходимо объявлять все сигналы, используемые в архитектуре, кроме портов. Но индексная переменная оператора **generate** не объявляется.

Также в области объявлений объявляются используемые компоненты, как, например, компонент LUT4, процедуры с их описаниями и т.п.

Поскольку при описании архитектуры используются только операторы вставки компонента, то такому описанию соответствует некоторая структура. Поэтому говорят, что архитектура описана структурным стилем. Результирующая структура, которая была построена компилятором-синтезатором XST, показана на рис.5.

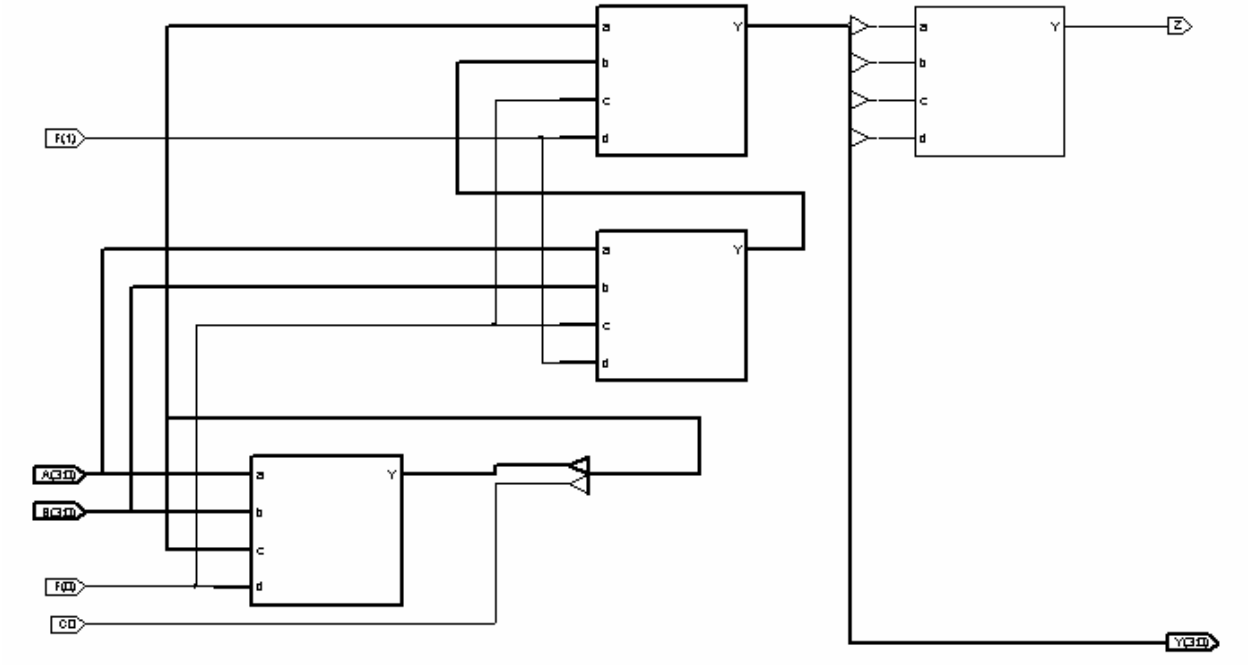


Рис.5. Структура LSM, спроектированного на базе LUT.

4.3. Структурная модель LSM на базе PLM.

Предыдущий пример LSM на базе LUT, можно перепроектировать с использованием элементов PLM вместо LUT. У всех PLM будет одинаковое объявление объекта:

```
use  Cnetlist.all;
entity PLM_4 is
 generic(td:time:=1 ns);            -- задержка
 port(A : in BIT;
      B : in BIT;
      C : in BIT;
      D : in BIT;
      Y : out BIT);
end PLM_4;
```

Для входной логической схемы LNI, согласно таблице истинности функции X_i (см. табл.1), архитектура выглядит следующим образом:

```
architecture PLMI of PLM_4 is
    begin
     Y<=(        -- арифметические операции
        (not D and not C and not B and a)        -- A+B
        or (not D and not C and B and not A)
        or (not D and  C and not B and not A)    -- A-B
        or (not D and  C and  B and A)
        --          логические операции
        or ( D and not C and B and A   )         -- AND
        or (( D and C) and not(not B and not A)) -- OR
        )           after td; --задержка элемента
    end PLMI;
```

Здесь принимается, что на входы D,C,B,A поступают переменные, соответственно F(1),F(0),B(i),A(i). Аналогично в соответствии с таблицами истинности описываются блоки LNO и LNS как архитектуры PLMO и PLMSб соответственно

```
architecture PLMO of PLM_4 is
    begin
     Y<=(        -- арифметические операции
        (not D and not C and not B and A) --     A+B
        or (not D and not C and B and not A)
        or (not D and  C and not B and not A)--  A-B
        or (not D and  C and  B and A)
                   --Логические операции
        or( D and not C and  B)       --  AND
        or( D and C and  B)           --  OR
        )           after td; --задержка элемента
    end PLMO;

architecture PLMS of PLM_4 is
    begin
     Y<=(        -- арифметические операции
        (not D and not C and  B and A)          -- A+B
        or (not D and C and not(not B and not A))
        or ( D and not C and not B and A)        --A-B
        or ( D and C and not B)
        or ( D and C and  B and A)
        )           after td; --задержка элемента
    end PLMS;
```

Элемент ИЛИ-НЕ, определяющий, что все разряды результата – нулевые, описывается следующей архитектурой

```
architecture PLM_NOR of PLM_4 is
    begin
     Y<=not (D or C or  B or A)  -- функция ИЛИ-НЕ
         after td;  --задержка элемента
    end PLM_NOR;
```

Теперь, когда архитектуры всех составляющих блоков подготовлены, можно составить описание архитектуры LSM в целом. Оно выглядит следующим образом.

```
    architecture STR_PLM of LSM is
     signal c,x,yi: BIT_VECTOR(4 downto 0); --внутренние сигналы
     component PLM_4 is    --используемый компонент
                           --с разными архитектурами
          generic(td:time:=1 ns);
          port(a : in BIT;
               b : in BIT;
               c : in BIT;
               d : in BIT;
               Y : out BIT);
     end  component;
```

```
begin
 c(0)<=C0;--входной перенос
-- 4 разряда LSM
 LSM_STR:for i in 0 to 3 generate
      LNI:entity PLM_4(PLMI) port map  -- блок LNI
          (a=>A(i),b=>B(i),c=> F(0),d =>F(1),
           Y =>X(i));
      LNO: entity PLM_4(PLMO) port map  -- блок LNO
          (a=>C(i),b=>X(i),c=> F(0),d =>F(1),
           Y =>yi(i));
      LNC:entity PLM_4(PLMS) port map   -- блок LNC
          (a=>A(i),b=>B(i),c=> c(i),d =>F(0),
           Y =>c(i+1));
 end generate;

 UZ:entity PLM_4(PLM_NOR) port map       --Элемент ИЛИ-НЕ
          (yi(3),yi(2),yi(1),yi(0),Z);

 Y<=yi(3 downto 0);                       -- Результат
 C3<=c(4);

end STR_PLM;
```

Следует отметить, что при вставке компонентов с архитектурами PLMI, PLMO, PLMS применено поименованное связывание сигналов и портов. Такое связывание предпочтительнее, т.к. в нем явно указано отношение сигнала и порта, что предупреждает появление ошибок. При вставке компонента – объекта PLM_4(PLM_NOR) применено позиционное связывание, т.е. присоединяемый сигнал ставится в позиции, отвечающей нужному порту. Такое связывание здесь уместно, т.к. функция ИЛИ-НЕ не зависит от перестановки входных переменных и поэтому ошибку при связывании выполнить трудно.

4.4. Испытательный стенд для LSM.

Вычислительные устройства обычно тестируются на всех этапах проектирования. Для автоматического тестирования VHDL-моделей используют, так называемый, испытательный стенд (testbench).

Один из способов тестирования заключается в сравнении тестируемой модели вычислителя с эталонной. Рассмотрим испытательный стенд для архитектуры LSM(STR_LUT), у которой эталонной моделью будет архитектура LSM(BEH).

Такой испытательный стенд – объект lsm_tb - представлен ниже.

```
entity lsm_tb is
end lsm_tb;
architecture TB_ARCHITECTURE of lsm_tb is
 component lsm --объявление тестируемого и эталон. объектов
      port(F : in BIT_VECTOR(1 downto 0);
           A : in BIT_VECTOR(3 downto 0);
           B : in BIT_VECTOR(3 downto 0);
           C0: in BIT;
           Y : out BIT_VECTOR(3 downto 0);
           C3: out BIT;
           Z : out BIT );
 end component;
 component  RANDOM_GEN is
      generic(n:positive:=4; --разрядность выходного слова
           tp:time:=100 ns     ;     -- период следования
           SEED:positive:=12345);    -- начальное состояние
      port(CLK:out BIT;
           Y : out BIT_vector(n-1 downto 0));
 end component;
--тестирующие сигналы
 signal F : BIT_VECTOR(1 downto 0):="00";
 signal C0 : BIT:='0';
 signal A,B : BIT_VECTOR(3 downto 0);
--проверяемые сигналы
 signal Y1,Y2,Y : BIT_VECTOR(3 downto 0);
 signal C31,C32,C, Z1,Z2,Z: BIT;
begin
 G1: RANDOM_GEN                    --генератор операнда A
     generic map(n=>4,SEED=>1234)
      port map(CLK=>open,Y =>A);
 G2: RANDOM_GEN                    --генератор операнда B
     generic map(n=>4,SEED=>8765)
     port map(CLK=>open,Y =>B);

 UUT1 :entity lsm(STR_LUT)         --тестируемый объект
     port map ( F => F,   A => A,   B => B,   C0 => C0,
      Y => Y1,  C3 => C31, Z => Z1);
 UUT2 :entity lsm(BEH)             --эталонный объект
      port map (     F => F,A => A, B => B,   C0 => C0,
      Y => Y2, C3 => C32, Z => Z2   );
--компараторы для сравнения результатов
 COMP_Y:   Y<=Y1 xor Y2;
 COMP_C:   C<=C31 xor C32;
 COMP_Z:   Z<=Z1 xor Z2;
end TB_ARCHITECTURE;
```

Объект `lsm_tb` не имеет портов ввода-вывода и поэтому у него простое объявление. Эталонный и тестируемый объекты LSM имеют одинаковый интерфейс, но различные архитектуры. Поэтому они объявлены одинаково, но при их вставке они обозначены как `entity lsm(BEH)` и `entity lsm(STR_LUT).` При вставке компонентов генераторов случайных чисел с помощью настроечных переменных были заданы разрядности их выходов n=4 и разные начальные состояния, чтобы выдаваемые ими значения операндов A и B были различными.

Структура испытательного стенда, построенная утилитой Code2Graphics системы Active HDL показана на рис. 6. На оба тестируемых компонента, обозначенных метками `UUT1` и `UUT2` подаются одинаковые операнды A и B с генераторов случайных чисел и управляющие сигналы Y и C0. Последние меняются вручную или в процессе

моделирования, или изменяя их начальное значения и перекомпилируя файл испытательного стенда.

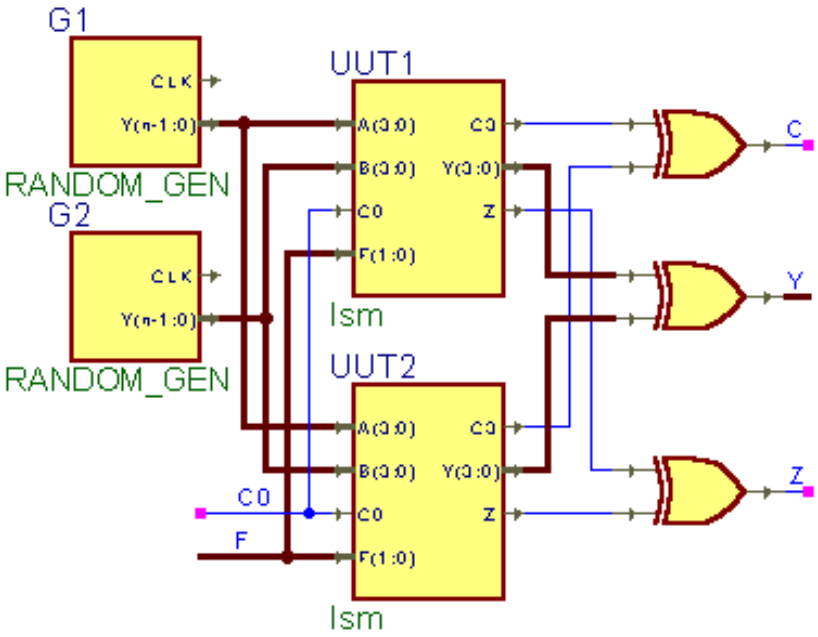


Рис.6. Структура испытательного стенда.

Результаты вычислений в LSM обоих типов сравниваются с помощью функций Исключающее ИЛИ. Если при моделировании результаты одинаковые, то эти фунции равны нулю.

5. **Порядок проведения лабораторной работы**

В соответствии с номером варианта, выбирается задание на выполнение лабораторной работы. Параметры задания включают:

- тип логического элемента (PLM или LUT);
- максимальное число термов PLM или количества входов LUT (4 или 5);
- разрядность результирующей схемы LSM;
- перечень операций LSM;
- перечень выходных сигналов.

Выполнение лабораторной работы имеет 3 стадии: разработка поведенческой модели LSM, разработка структурной модели LSM и разработка испытательного стенда с проверкой функционирования LSM.

5.1. **Разработка поведенческой модели LSM**.

Поведенческая модель LSM описывается стилем потоков данных с использованием операций с целыми числами и функций из пакета `Cnetlist` . При этом используется редактор и компилятор VHDL, входящие в состав пакета Active HDL.

После того, как LSM описана в виде архитектуры LSM(BEH), она тестируется путем ручной подачи входных тестовых значений при моделировании этой архитектуры.

5.2. **Разработка структурной модели LSM.**

Структурная поведенческая модель LSM описывается структурным стилем. При этом используются компоненты из файла `Cnetlist_Lib.VHD` . Для этого также используется редактор и компилятор VHDL, входящие в состав пакета Active HDL.

5.3. **Разработка испытательного стенда и тестирование моделей.**

За образец испытательного стенда берется его пример, описанный в п.4.3. Он дорабатывается под требования конкретного испытуемого объекта.

При тестировании моделей по графикам сигналов определяется правильность функционирования моделей и измеряются задержки сигналов между входами и выходами структурной модели. Полученные графики сигналов переносятся в файл отчета с помощью функций выделения и сохранения в "кармане". По результатам тестирования формулируются выводы по лабораторной работе.

6. Отчет по лабораторной работе.

Отчет по лабораторной работе должен содержать:

- цель работы,
- описание варианта LSM,
- ход синтеза моделей PLM или содержимого LUT,
- тексты описаний поведенческой и структурной моделей LSM,
- графики сигналов, снятых на испытательном стенде,
- измеренные задержки сигналов,
- выводы.

7. Вопросы по лабораторной работе.

Каково функциональное назначение LSM?

Каким образом управляют режимом работы LSM?

Какой принцип функционирования базовых элементов, применяемых в ПЛИС и CPLD?

Для чего нужен пакет в VHDL?

Назначение объявления объекта в VHDL.

Для чего нужны переменные generic в VHDL?

Из каких частей состоит описание объекта в VHDL?

Какова структура описания архитектуры в VHDL?

Как объявляются векторы бит?

Как к проекту на VHDL подключают библиотеки?

Что такое программирование стилем потоков данных?

Как выполнить вставку компонента в VHDL?

Какое различие между позиционным и поименованным связываниями?

Что такое программирование структурным стилем?

Что такое программирование стилем для синтеза?

Чем отличаются параллельные операторы от последовательных?

Какие различия между сигналом и переменной в VHDL?

Каковы функции испытательного стенда в VHDL?

Что такое поведенческая модель в VHDL?

Почему логические схемы можно описывать стилем потоков данных, а схемы с регистрами – нельзя?

В какие аппаратные элементы отображаются функции преобразования типа?

Что выполняет функция конкатенации в VHDL?