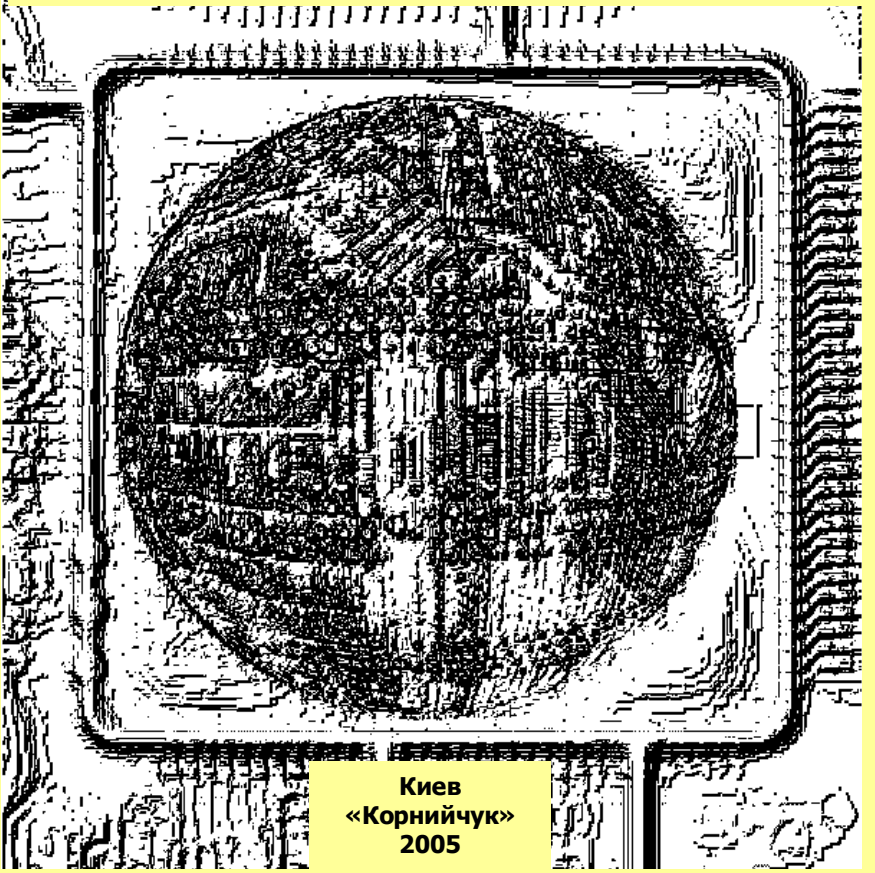
Сергиенко А.М., Корнейчук В.И.

Микропроцессорные устройства на программируемых логических ИС



Киев
«Корнийчук»
2005

УДК 681.3
ББК 32.973-01
С32

Р е ц е н з е н т : А.М.Романкевич, доктор технических наук, профессор (кафедра специализированных вычислительных систем Национального технического университета Украины „КПИ")

Сергиенко А. М., Корнейчук В.И.
С32 Микропроцессорные устройства на программируемых логических ИС. –К.: «Корнійчук», 2005. -108 с.
ISBN 966-7992-23-3

Изложены основы языка VHDL, используемого для проектирования современных вычислительных средств. Описаны основные особенности применения программируемых логических ИС при проектировании на VHDL а также соответствующие инструментальные средства проектирования. На основе серии лабораторных работ описывается ход проектирования на VHDL микропроцессорных устройств.

Для студентов, аспирантов, преподавателей вузов и специалистов по разработке аппаратных средств вычислительной, управляющей и измерительной техники.

ББК 32.973-01

Сергієнко А. М., Корнійчук В.І.
С32 Мікропроцесорні пристрої на програмованих логічних ІС. –К.: «Корнійчук», 2005. -108 с.
ISBN 966-7992-23-3

Викладені основи мови VHDL, що застосовується для проекттування сучасних обчислювальних засобів. Описані основні особивості застосування програмованих логічних ІС при проектуванні на VHDL а також відповідні інструментальні засоби проектування. На основі серії лабораторних робіт описано хід проектування на VHDL мікропроцесорних пристроїв.

Для студентів, аспирантів, викладачів вузів і спеціалістів по розробці аппаратних засобів обчислювальної, керуючої і вимірювальної техніки.

ББК 32.973-01
ISBN 966-7992-23-3

Сергиенко А.М., Корнейчук В.И.

Микропроцессорные устройства на программируемых логических ИС

Киев
«Корнійчук»
2005

А.М.СЕРГИЕНКО, В.И.КОРНЕЙЧУК
МИКРОПРОЦЕССОРНЫЕ УСТРОЙСТВА НА ПРОГРАММИРУЕМЫХ ЛОГИЧЕСКИХ ИС

Содержание

ВВЕДЕНИЕ 3
1. ОСНОВЫ ЯЗЫКА VHDL 5
Общая характеристика 5
Основные конструкции языка 6
Типы данных, объекты и выражения языка 13
Процессы и последовательные операторы 17
Подпрограммы 20
Параллельные операторы 22
Атрибуты 24
Пакеты для проектирования вычислительных устройств 26
2. ЭЛЕМЕНТНАЯ БАЗА МИКРОСХЕМ С ПРОГРАММИРУЕМОЙ ЛОГИКОЙ 29
Общая характеристика 29
Логический элемент ПЛМ 29
Логический элемент ПЛИС 30
Триггеры в ПЛМ и ПЛИС 32
Блоки памяти в ПЛИС 34
Тристабильные схемы в ПЛИС и CPLD 37
Пакет CNetwork 38
3. ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ПРОЕКТИРОВАНИЯ 41
Система Active HDL 41
Система проектирования ПЛИС 45
4. ЛАБОРАТОРНЫЕ РАБОТЫ 48
Общие требования к лабораторным работам 48
Арифметико-логическое устройство 50
Счетчик команд 60
Оперативное запоминающее устройство 70
Регистровая память 76
Блок умножения 80
Арифметическое устройство 87
Ядро микропроцессора 95
ЛИТЕРАТУРА 107
ПРИЛОЖЕНИЕ 108

ВВЕДЕНИЕ

Микропроцессорная элементная база стала аппаратной основой практически всей вычислительной техники в конце прошлого века. Микропроцессор – это вычислительный модуль, обладающий уникальными свойствами широчайшей универсальности и настраиваемости под решение конкретных вычислительных задач. Эта универсальность достигается благодаря тому, что в рамках архитектурных особенностей микропроцессора можно реализовать любой вычислительный алгоритм путем его программирования.

Так как разработка и внедрение новых микропроцессоров, включая их системное матобеспечение, исключительно высокозатратны, разработка и производство новых микропроцессоров были возможными, в основном, только на крупных фирмах с многолетним опытом работы в этой отрасли. Поэтому номенклатура архитектурных групп микропроцессоров ограничивалась несколькими десятками.

Любую сложную операцию в микропроцессоре можно вычислить путем исполнения последовательности команд определенной длины, т.е. с помощью подпрограммы. Современные вычислительные задачи, такие как задачи мультимедиа, распознавания образов, кодирования информации, требуют выполнения большого числа сложных операций в реальном масштабе времени. Это сводится к повторению исполнения команд в подпрограммах с частотой более миллиарда команд в секунду. При этом ускорение выполнения этих задач уже не может быть достигнуто путем повышения тактовой частоты микропроцессоров, которая сейчас приближается к физической границе для кремниевой технологии. Кроме того, такое увеличение тактовой частоты влечет за собой энергопотребление микропроцессоров, превосходящее десятки ватт, что неприемлемо для портативных устройств.

Для решения современных вычислительных задач создаются новые микропроцессоры, которые благодаря адаптации своей архитектуры к особенностям этих задач, решают их в реальном масштабе времени с минимизированным энергопотреблением. Это, в основном, микропроцессоры встраиваемого применения или микроконтроллеры. Современный микропроцессор, кроме своего ядра, включает в себя набор периферийных устройств, оперативную и постоянную память достаточного объема, аппаратные ускорители, средства ввода-вывода аналоговых сигналов, объединенные между собой стандартными интерфейсными шинами, сочетание которых определяется особенностями конкретной области применения. Такой микропроцессор или система из них, реализованные на одной микросхеме, принято называть системой на кристалле (System-On-the-Chip, SOC).

Номенклатура и сложность SOC постоянно растут, а сроки их проектирования остаются ограниченными требованиями скорейшего их внедрения. Основой технологии проектирования SOC остается описание вычислительных блоков на уровне регистровых передач (Register Transfer Level – RTL) с помощью языков описания аппаратуры VHDL и Verilog с последующей автоматической трансляцией до уровня логических схем и далее – до уровня технологической документации. Для ускорения процесса проектирования внедряются три новых направления в проектировании: повторное

использование вычислительных модулей (IP Cores), совместное аппаратно-программное проектирование микропроцессорных устройств и системный синтез вычислительных устройств.

Для сокращения сроков и затрат на разработку и внедрение новых вычислительных средств, таких как SOC, все чаще используются программмируемые логические интегральные схемы (ПЛИС или FPGA). Современные ПЛИС предоставляют разработчикам вычислительных устройств программируемые аппаратные ресурсы объемом от тысяч до миллионов вентилей. Проектирование устройства на ПЛИС сводится к его описанию на RTL – уровне и к автоматической трансляции этого описания в файл конфигурации ПЛИС. Причем такое проектирование не требует больших временных и трудовых затрат. Поэтому ПЛИС широко используется как элементная база для малосерийного производства вычислительных устройств.

Одно и то же RTL-описание устройства может быть автоматически транслировано как в конфигурацию ПЛИС, так и в комплект масок для заказной микросхемы. Изготовление опытной партии SOC слишком дорого. Поэтому, как правило, проектирование SOC сопровождается изготовлением и отладкой ее прототипа на базе ПЛИС со стремлением дальнейшего получения исправных кристаллов SOC с первого раза.

Таким образом, сегодня множество архитектур микропроцессоров, реализованных в SOC, стремительно расширяется. ПЛИС становятся неотъемлемым атрибутом технологии встраиваемых вычислительных средств. Растет потребность в инженерах-проектировщиках, владеющих технологиями проектирования SOC. От них требуется умение описывать на VHDL или Verilog устройства с заданной функциональностью, включая микропроцессор, переналаживать готовые блоки под требования проекта, соединять модули в структуру SOC и программировать встроенное матобеспечение.

В книге предлагается цикл лабораторных работ по изучению компьютерной схемотехники. Основу лабораторных работ составляет проектирование узлов микропроцессоров с использованием языка VHDL и соответствующих средств проектирования, включая симулятор VHDL и компилятор-синтезатор логических схем. Специалист, выполняющий курс лабораторных работ, изучает не только основы схемотехники микропроцессоров, но и современную технологию их проектирования для реализации в ПЛИС.

В первой главе изложены основы языка VHDL. Во второй и третьей главе приведены архитектурные особенности ПЛИС и описание средств их проектирования. В четвертой главе описаны лабораторные работы. Каждая следующая лабораторная работа отличается нарастающей сложностью и использует результаты предыдущих лабораторных работ. В заключительной лабораторной работе выполняется сборка и испытание модели учебного микропроцессора, состоящего из модулей, спроектированных в предыдущих лабораторных работах. Поэтому лабораторные работы рекомендуется выполнять последовательно, начиная с первой и без пропусков.

Хотя проекты устройств, разрабатываемых при выполнении лабораторных работ, являются учебными, их сущность и ход их проектирования соответствуют практике проектирования реальных устройств.

ОСНОВЫ ЯЗЫКА VHDL

1.1 Общая характеристика

По ряду признаков язык VHDL является производным от языка Ada. У этих языков почти одинаковый синтаксис, множества операций и операторов. Так, например, комментарии в обоих языках начинаются двойным тире и оканчиваются в конце строки. Для практики программирования важно то, что VHDL также как и Ada - строго типизированный язык.

Вычисления, запрограммированные на VHDL, основываются на множестве процессов, выполняемых параллельно и обменивающихся сигналами. Сигналы играют роль как носителей информации между процессами, так и синхронизирующих воздействий, запускающих процессы на исполнение. Поэтому как и Ada, VHDL – это язык параллельного программмирования [1]. Специалисту, знакомому с Ada, легко программировать на VHDL. При программировании на VHDL почти никогда не используются ресурсы, разделяемые между несколькими процессами. Благодаря этому, исчезает сложная проблема синхронизации доступа к таким ресурсам и в результате, VHDL – программы исполняются значительно быстрее, чем программы на Ada, а число процессов может достигать десятков и сотен тысяч.

Отличием VHDL является то, что типы и объекты языка изначально имеют прямую аналогию с физическими сущностями и элементами дискретной электроники. Так, введен класс физических типов, основным из которых является время – time. В отличие от числовых типов, физический тип имеет конкретные единицы измерения. Например, тип time обозначается в секундах и стандартных их долях, т.е. в наносекундах, микросекундах и т.п. Проектирование вычислительных средств на VHDL основано на том, что под сигналами подразумеваются реальные линии связи в устройствах, а под процессами – конкретные элементы и узлы компьютеров, поведение которых совпадает с поведением этих процессов. Если всем процессам и сигналам в VHDL – программе находится полная аналогия с линиями связи, элементами и узлами некоторого вычислительного устройства, то говорят, что такая программа написана стилем для синтеза. Будучи запущенной на исполнение, эта программа ведет себя с большой точностью так же, как и реальное устройство. Процессы обмениваются сигналами в те же моменты времени и в том же порядке, что и узлы в подразумеваемом устройстве.

Существуют компиляторы-синтезаторы, которые выполняют семантический анализ программ, написанных стилем для синтеза, и затем во взаимно однозначном соответствии строят искомую логическую схему в заданном элементном базисе. Одновременно с этим выполняется минимизация булевских выражений и результирующих аппаратных ресурсов и оптимизация быстродействия, энергопотребления, проверка правил проектирования и т.п.

1.2 Основные конструкции языка

Дискретная система может быть представлена в VHDL как объект проекта. Объект – это основная составная часть проекта. Объект может быть использован в другом проекте, который, в свою очередь, описан как объект или может являться объектом более высокого уровня в данном проекте.

Объект проекта описывается набором составных частей проекта, таких как: объявление объекта, тело архитектуры объекта (или просто архитектура), объявление пакета, тело пакета и объявление конфигурации. Каждая из составных частей объекта может быть скомпилирована отдельно. Составные части проекта сохраняются в одном или нескольких текстовых файлах с расширением .VHD. В одном файле может сохраняться несколько объектов проекта. Объект проекта обычно описывается согласно синтаксису:

```
\объект проекта\::= [\описание library\]
                    [\описание use\]
                   \объявление объекта\
                    \тело архитектуры\
                  [\объявление конфигурации\]
```

где идентификаторы \описание **library**\– названия библиотек, которые используются в объекте проекта и указывают транслятору местоположение этих библиотек. Описание **use** указывает, какие пакеты и какие элементы этих пакетов могут быть использованы в объекте.

Здесь и далее жирным шрифтом выделены ключевые слова языка. Слово в обратных косых означает синтаксическую единицу языка и представляет собой расширенный идентификатор, который может содержать любые символы. Фраза, заключенная в квадратные скобки, может быть опущена, а в фигурных скобках – может повторяться несколько раз.

Объявление объекта.

Объявление объекта указывает, как объект проекта выглядит снаружи и каким образом его можно включить в другом объекте проекта в качестве компонента, т.е. он описывает внешний интерфейс объекта. Упрощенный синтаксис объявления объекта:

```
\объявление объекта\::= entity \идентификатор\ is
        [generic(\объявление настроечной константы\
                {; \объявление настроечной константы\});]
        [port (\объявление порта\  {;\объявление порта\});]
    end \идентификатор\;
```

Здесь \идентификатор\ - имя объекта. Объявление портов, обозначенное ключевым словом **port**, представляет собой набор интерфейсных сигналов объекта проекта. Настроечные константы **generic** кодируют определенные свойства объекта проекта, например, разрядность линий связи, параметры задержки, кодирование структуры моделируемого устройства. Следует учитывать, что в VHDL идентификаторы с одинаковым написанием представляют один и тот же объект, т.е. они «нечувствительны» к высоте букв.

На рис.1.1 показано изображение объекта 2-разрядного сумматора, соответствующего следующему объявлению объекта

```
entity SM2 is
generic(td:time:=1 ns);
port (C0:in bit;   -- синхросигнал
  A:in bit_vector(1 downto 0);  --входное данное
  B:in bit_vector(1 downto 0);  --входное данное
  C2: out bit; -- выход переноса
  Q:out bit_vector(1 downto 0));--выходное данное
end SM2;
```

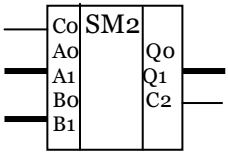


Рис.1.1. Объект двухразрядного сумматора

Здесь вход и выход переноса Co,C2 имеют битовый тип, а вход D и выход Q – тип вектора битов длиной 2. Ключевые слова **in**, **out** обозначают направление передачи информации, соответственно, в объект и из объекта. Спадающий диапазон 1 **downto** 0 указывает порядок нумерации битов в векторе – справа – налево. Нумерация слева – направо задается как 0 **to** 1. Настроечная константа td типа time задает задержку каскадов сумматора.

Архитектура объекта.

Архитектура объекта представляет собой отдельную часть, в которой описано, каким образом реализован объект. Ее упрощенный синтаксис:

```
\тело архитектуры\::= architecture \идентификатор\ of \имя объекта\ is
                        {\объявление в архитектуре\}
                      begin
                    { \параллельный оператор\}
                      end [architecture][\идентификатор\];
```

Идентификатором обозначается имя конкретного тела архитектуры, а имя объекта указывает, который из объектов описан в этом теле архитектуры. Одному объекту проекта может соответствовать несколько архитектур, в каждой из которых описан один из вариантов реализации объекта. Объявленные в теле архитектуры типы, сигналы, подпрограммы видимы только в пределах этой архитектуры.

Исполнительную часть архитектуры составляют параллельные операторы, такие как процесс, параллельное присваивание сигналу, вставка компонента и др. Так как все операторы в исполнительной части тела архитектуры – параллельные, их взаимный порядок – безразличен.

Существуют три основных стиля программирования архитектур объекта: поведенческий, структурный стили и стиль потоков данных. Часто программирование выполняется с комбинированием этих стилей.

Поведенческий стиль

В основу языка VHDL положены операторы процесса. Каждый такой оператор - это описание поведения некоторого виртуального процессорного элемента в зависимости от приходящих в него сигналов и состояния пере-

менных и сигналов, изменяемых и сохраняемых в нем. Т.е. операторы процесса задают поведение сигналов в зависимости от событий, происходящих во времени. Поэтому если в архитектуре только операторы процесса, то говорят, что такая архитектура описана поведенческим стилем.
Одна из большого множества возможных архитектур двухразрядного сумматора, описанного поведенческим стилем, выглядит так:

```
architecture BEHAVIORAL of SM2 is
begin
        SM: process(C0,A,B)
                variable S:integer; -- счетчик единиц во входных данных
        begin
                S:=0;   -- установка в 0 счетчика
                if C0='1' then S:=S+1; end if;  -- инкремент счетчика на 1, если
                if A(0)='1' then S:=S+1; end if; -- в младших разрядах 1
                if B(0)='1' then S:=S+1; end if;
                if A(1)='1' then S:=S+2; end if; -- инкремент счетчика на 2, если
                if B(1)='1' then S:=S+2; end if;        -- в старших разрядах 1
                if S mod 2 = 1 then    -- выделение младшего разряда суммы
                        Q(0)<='1' after td;
                else
                        Q(0)<='0' after td;
                end if;
                if (S/2) mod 2 = 1 then  -- выделение старшего разряда суммы
                        Q(1)<='1' after 2*td;
                else
                        Q(1)<='0' after 2*td;
                end if;
                if S>3 then    -- выделение разряда переноса
                        C2<='1'  after 2*td;
                else
                        C2<='0'  after 2*td;
                end if;
        end process;
    end BEHAVIORAL;
```

Сигналы C0,A,B, заключенные в скобки, представляют собой список чувствительности процесса. Тело оператора процесса образует цепочка последовательных операторов, в данном случае – это условные операторы **if-then-else** . Вначале выполнения программы – запуске моделирования архитектуры в симуляторе – сигналам и переменным присваивается некоторое начальное значение. Если это значение не задано, то для битового типа оно равно нулю. Далее выполняются операторы процесса, и он останавливается после исполнения последнего оператора.

Процесс запускается по каждому событию изменения любого из сигналов из списка чувствительности. После каждого запуска последовательно исполняются операторы тела процесса, и он останавливается за последним оператором. Здесь при каждом запуске процесса переменная S устанавливается в 0, затем она последовательно увеличивается на 1, если входные сигналы переноса или младших разрядов операндов равны 1 и увеличивается

на 2, если старшие разряды операндов – единичные. Затем выходным сигналам присваивается соответствующее значение в зависимости от подсчитанной величины S. При этом присваивание сигналу выполняется с задержкой на заданную величину td или 2*td, которая указывается словом **after**.

Присваивание переменной выполняется оператором ":=", а присваивание сигналу – оператором "<=". Присваивание переменной выполняется сразу же при выполнении этого оператора, как в обычных языках программирования. По оператору "<=" сначала выполняется назначение сигналу, а собственно присваивание с отработкой задержки, если есть слово **after**, выполняется после остановки процесса.

Стиль потоков данных

Параллельные операторы присваивания, условного и селективного присваивания указывают потоки данных между линиями связи, обозначенными идентификаторами сигналов, а также обработку этих потоков. Поэтому если в теле архитектуры встречаются только такие операторы, то говорят, что эта архитектура написана стилем потоков данных. Пример архитектуры того же сумматора, описанной стилем потоков данных:

```
architecture DATAFLOW of SM2 is
 signal C1: bit;
begin
Q(0) <= A(0) xor B(0) xor C0 after td;
C1   <= (A(0) and B(0)) or (A(0) and C0) or (B(0) and C0) after td;
Q(1)<= A(1) xor B(1) xor C1 after td;
C2   <= (A(1) and B(1)) or (A(1) and C1) or (B(1) and C1) after td;
end DATAFLOW;
```

Здесь логика работы сумматора задана логическими выражениями с параллельным присваиванием результатов сигналам. Нельзя забывать, что логические операторы **xor**, **and**, **or** – равноприоритетны. Поэтому порядок их применения необходимо задавать скобками. Эти операторы применимы также к битовым векторам. Например, тело этой архитектуры может выглядеть так:

```
Q  <=A xor B xor C1&C0 after td;
CB<=(A and B) or (A and C1&C0) or (B and C1&C0) after td;
C2<=CB(1);    C1<=CB(0);
```

Здесь функция "&" означает объединение (конкатенацию) бит или векторов в один вектор, - в данном случае - в вектор переносов.

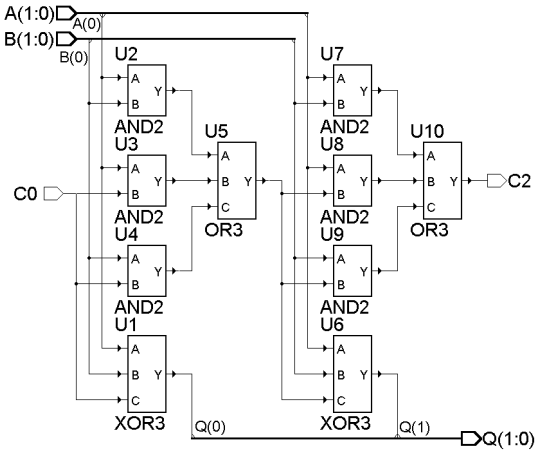


Рис.1.2. Структура двухразрядного сумматора

Поскольку все операторы – параллельные, то их порядок в программе – безразличен. Но, в отличие от присваивания в процессе, здесь нельзя дважды присваивать какое-либо выражение одному и тому же сигналу.

Структурный стиль

Структурное описание системы задает компоненты проекта и их соединения. Алгоритм поведения такого проекта не задан явным образом, но он выражен в структурном виде. Ниже приведен пример описания двухразрядного сумматора, структура которого показана на рис.1.2.

```
architecture STRUCT of SM2 is
component AND2  port(A,B: in bit; Y: out bit);
end component;
component OR3   generic(TD:time);  port(A,B,C: in bit; Y: out bit);
end component;
component XOR3   generic(TD:time);  port(A,B,C: in bit; Y: out bit);
end component;
signal C1: bit; --бит переноса с разряда 0 в разряд1
signal t1,t2,t3,t4,t5,t6: bit; --промежуточные биты
begin
    U1:XOR3 generic map(td) port map(A(0),B(0),C0,Q(0));
    U2:AND2 port map(A(0),B(0),t1);
    U3:AND2 port map(A(0),C0,t2);
    U4:AND2 port map(B(0),C0,t3);
    U5:OR3   generic map(td) port map(t1,t2,t3,C1);
    U6:XOR3 generic map(td) port map(A(1),B(1),C1,Q(1));
    U7:AND2 port map(A(1),B(1),t4);
    U8:AND2 port map(A(1),C1,t5);
    U9:AND2 port map(B(1),C1,t6);
    U10:OR3 generic map(td) port map(t4,t5,t6,C2);
 end STRUCT;
```

В декларативной части архитектуры объявлены компоненты, которые используются в описываемой структуре. Эти компоненты могут быть библиотечными компонентами или описаны в другом месте, например, как следующие объекты.

```
    entity AND2 is port (A,B: in bit; Y: out bit);
end AND2;
architecture beh of AND2 is begin
   Y<=A and B;
end beh;
entity XOR3 is generic(TD:time);  port (A,B,C: in bit; Y: out bit);
end XOR3;
architecture beh of XOR3 is
begin
   Y<=A xor B xor C after TD;
end beh;
entity OR3 is generic(TD:time);  port (A,B,C: in bit; Y: out bit);
end OR3;
architecture beh of OR3 is
begin
   Y<=A or B or C after TD;
end beh;
```

Операторы, отмеченные метками U1,...,U10 – это операторы вставки компонента. Поскольку это параллельные операторы, то их порядок в программе безразличен. Во фразе связывания настроечных констант **generic map**(td) выполняется передача настроечной константы задержки элемента. Во фразе связывания портов и сигналов **port map** сигналы и порты компонентов соединяются в соответствии с графом структуры. Здесь выполнено ассоциативное связывание, когда сигнал соотносится с портом в том же порядке, который задан в объявлении компонента. Но чаще встречается поименованное связывание, когда сигнал относится к порту явным образом и порядок связываний портов безразличен, как, например:

```
U2:AND2 port map(Y=>t1,A=>A(0),B=>B(0));
```

Пакеты

Пакет – это набор объявлений типов, констант, подпрограмм, компонент и т.д., которые оформлены таким образом, чтобы быть доступными для различных проектов и использоваться ими сообща. Интерфейс пакета выполнен в виде объявления пакета в соответствии с синтаксисом:

```
\объявление пакета\::= package \идентификатор\ is
                       {объявление в пакете}
                        end [package][\идентификатор\];
```

С другой стороны, тело пакета содержит скрытые детали, которые невидимы для пользователей пакета и имеет следующий синтаксис:

```
\тело пакета\::= package body \идентификатор\ is
                       {объявление в теле пакета}
                            end [package body][\идентификатор\];
```

В примере поведенческого описания несколько логических операторов применялись для преобразования целого операнда в вектор бит. Целее-

сообразно составить функцию преобразования целого в вектор бит и использовать ее в нескольких проектах, оформив в виде следующего пакета.

```
Package INT_BIT is
  constant GND : bit:='0';
  constant ONE : bit:='1';
        --преобразование целого со знаком в вектор бит - число в прямом коде
  function INT_TO_BIT(arg, -- целое положительное
                        len:integer --длина вектора, не более 32
                           ) return bit_vector;
end INT_BIT;
Package body INT_BIT is
function INT_TO_BIT(arg,len:integer) return bit_vector is
        variable bv: bit_vector(len-1 downto 0);
        variable ii,i2:integer;
begin
        ii:=arg;
        for j in 0 to len-1  loop -- цикл перевода целого в двоичное число
                i2:=ii/2;
                if i2*2/=ii then
                        bv(j):='1';
             else
                        bv(j):='0';
                end if;
                ii:=i2;
        end loop;
        return bv;
end function;
end package body;
```

При вставке компонентов, некоторые их входы могут не использоваться. Поэтому заданные в пакете константы GND и ONE удобно применять для подачи определенных сигналов на неиспользуемые входы. При опиисании функции используются такие элементы языка, как оператор цикла и оператор **return**. Оператор цикла **for** j **in** 0 **to** len-1 **loop** выполняет len раз тело цикла, причем переменная цикла j пробегает значения от 0 до len-1. Функция INT_TO_BIT возвращает свое значение по оператору **return**.
Для использования объектов, объявленных в конкретном пакете используют их селективные имена, как, например:

```
A:=INT_BIT.INT_TO_BIT(12,4);
```

Чтобы сделать эту функцию видимой для всего объекта проекта, перед ним ставят описание **use**:

```
use INT_BIT.INT_TO_BIT;
```

А если необходимо, чтобы всё содержимое библиотеки стало доступным, то применяют резервированное слово **all**:

```
use INT_BIT.all;
```

Стандарт языка VHDL включает в себя предопределенные пакеты: STANDARD и TEXTIO. Разработка других пакетов – это эффективный путь для создания стандартного или специального окружения при проектировании устройств на основе VHDL.
В стандартном пакете STD_LOGIC_1164 из библиотеки IEEE определен тип девятизначной логики и функции над этим типом, которые используются практически во всех проектах новых устройств вычислительной техники. Такие пакеты из библиотеки IEEE, как STD_LOGIC_ARITH, NUMERIC_STD, используются при синтезе устройств с арифметическими функциями. В данном цикле лабораторных работ широко используется пакет NUMERIC_BIT из той же библиотеки, содержащий функции над битовым типом данных.

1.3 Типы данных, объекты и выражения языка

То, что VHDL – это строго типизированный язык, означает, что каждый объект языка имеет тип, который задает, какое значение может быть присвоено объекту и какие операции могут с ним выполняться. Во время компиляции проверяются соответствия типа объекта и типов аргументов операций. При исполнении программ проверяется соответствие типов, назначаемых или уточняемых динамически, например, равенство длин векторов при присваивании. В пакете STANDARD заданы предопределенные типы: bit, bit_vector, integer, boolean, character, string, time. Другие типы определены в пакетах библиотеки IEEE или могут определяться пользователем.

Типы данных

В VHDL используются четыре класса типов данных: скалярные, композитные типы, типы доступа к динамической памяти и файловые типы.
Скалярные типы имеют простые и одиночные значения, присваиваемые из упорядоченного множества, т.е. с объектами скалярного типа можно выполнять операции сравнения. **Перечисляемый тип** определяется как список (перечисление) всех возможных значений данного типа. Например, объявление перечисляемого типа STD_ULOGIC из пакета STD_LOGIC_1164 выглядит так:

```
type STD_ULOGIC is ( 'U',  -- Неинициализировано
                     'X',  -- Сильное неизвестное
                     '0',  -- Сильный  0
                     '1',  -- Сильная  1
                     'Z',  -- Высокий импеданс
                     'W',  -- Слабое неизвестное
                     'L',  -- Слабый     0
                     'H',  -- Слабая     1
                      '-'   -- Безразлично                );
```

Целочисленный тип определяет множество значений целых чисел в заданном диапазоне:

```
type ADDRESS is range 0 to 65535;
```

Тип integer представляет собой предопределенный целочисленный тип в диапазоне от $-2^{31}+1$ до $2^{31}-1$.

Тип с плавающей запятой - это машинное приближение действительных чисел. Он определяется аналогично целому типу, но с диапазоном с плавающей запятой. Максимальный диапазон зависит от компилятора.

Физический тип - это числа, выражающие физические величины, такие как время, длина, напряжение и т.п. Чаще всего используется предопределенный тип time, задающий задержки сигналов, отсчет времени моделирования. Значения этого типа отсчитываются как целые числа в фемтасекундах, fs, пикосекундах, ps, наносекундах, ns, микросекундах, us и т.д.

Составные типы используются для определения наборов значений. В языке существуют регулярный тип и комбинированный тип.

Регулярный тип представляет собой множество элементов одинакового типа. Различают неограниченные и ограниченные регулярные типы. Неограниченный тип массива чисел с плавающей запятой:

```
type REAL_ARR is array (integer range<>) of real;
```

Выражение integer **range**<> означает, что у массива чисел диапазон индексов выражается целыми числами, но здесь он не определен. Этот диапазон должен быть определен потом при объявлении объектов или во время компиляции и связывании объектов, например, при вызове функции.
Примеры объявления ограниченного регулярного типа:

```
type MY_STRING is array (0 to 79) of character;
type ARR_4K is array (0 to 4095) of bit_vector(7 downto 0);
type ARR_4Kx8 is array (0 to 4095, 7 downto 0) of bit;
```

В первом случае задан тип строки стандартной длины 80, а во втором – тип массива байтов объемом 4К, который будет использоваться, например, для построения четырехкилобайтного ОЗУ. В третьем примере объявлен тип массива также размером 4 килобайта, но здесь массив двумерный и состоит из битов.

Комбинированный тип определяет множество значений, как и регулярный тип, но эти значения могут быть разнотипными. Пример объявления комбинированного типа:

```
type INSTR32 is record
 CODE:bit_vector(7 downto 0);
ADDR1:bit_vector(11 downto 0);
ADDR2:bit_vector(11 downto 0);
end record;
```

Он может быть использован для задания формата некоторой команды, имеющей поля кода операции CODE и адресов операндов ADDR1, ADDR2.

Подтип.

Подтипом называется тип с дополнительными ограничениями. Подтип используют для отождествления группы объектов базового или родоначального типа. В примерах объявления подтипов:

```
subtype Z01_U is STD_ULOGIC range '0' to 'Z';
subtype BYTE_I is integer range -128 to 127;
```

В первом случае задается подтип, состоящий из элементов '0', '1' и 'Z', а во втором случае – подтип целых чисел, которые представляют байты.

Ограничение подтипа позволяет выявить ошибки на этапе моделирования, когда ошибочный выход за границы подтипа фиксируется симулятором. Объекты разных подтипов, у которых один родоначальный тип, могут участвовать в вычислениях без конфликтов типов.

Подтипы natural и positive – это предопределенные подтипы, означающие целые положительные числа и целые числа больше нуля, соответственно.

Объекты языка

Объект языка VHDL - это программная единица с именем, которой можно присваивать значение определенного типа или подтипа. В VHDL используются четыре класса объектов: константы, сигналы, переменные и файлы. В момент своего создания **константа** инициализируется с присвоением заданного значения, которое не может быть изменено в процессе выполнения программы. Например, при объявлении константы:

```
constant MINUS1: bit_vector(31 downto 0):=(31=>'1', others =>'0');
```

константе присваивается агрегат (31=>'1', **others** =>'0'), который самому левому биту 32-разрядного вектора дает значение 1, а остальным битам (**others**) – значение 0.

Агрегатом называется операция, которая объединяет одно или несколько значений в значение составного типа.

Переменная создается при своем объявлении. Присваивание переменной происходит сразу же при выполнении оператора присваивания. Переменную можно объявлять только в операторах процесса и в описаниях подпрограмм. Поэтому переменные не имеют доступа от других процессов и подпрограмм. Это обязательное условие детерминированного поведения VHDL-моделей вычислительных устройств. При объявлении переменной ей можно присваивать начальное значение, как, например:

```
variable ct2: integer range 0 to 1023:=0;
```

Сигналы используются для соединения частей проекта в целое. Поэтому сигнал – это объект, с помощью которого информация передается между процессами и компонентами. Сигнал может быть запомнен в своей истории и воспроизведен в симуляторе в виде графика или таблицы. Объявление сигнала выглядит так же, как объявление переменной:

```
signal ct2: bit_vector(0 to 23);
```

Сигнал нельзя объявить внутри процесса или подпрограммы. Порты объекта проекта неявно представляют собой сигналы. Если сигнал или переменная не имеют явно заданного начального значения, то симулятор присваивает им начальное значение как самое левое значение из множества значений его типа. Например, если сигнал целого типа без диапазона, то его начальное значение будет $-2^{31}+1$.

Выражения

Выражения в VHDL представляют собой формулы, определяющие вычисления значений. В выражениях скомбинированы операции с объектами, литералами, вызовами функций и выражениями в скобках. Следующие операции могут быть использованы в выражениях. Они приведены в порядке своего приоритета.

Логические операции and, **or**, **nand**, **nor**, **xor**, **xnor**, **not**. Эти операции определены для предопределенных типов bit и boolean, а также для одномерных массивов элементов типа bit и boolean.

Операции сравнения = | /= | < | <= | > | >= . Равенство и неравенство определены для любого типа, кроме файлового. Остальные операции сравнения определены для любого скалярного типа или для одномерного регулярного типа. В последнем случае сравнение выполняется поэлементно, начиная с крайнего левого элемента.

Операторы сдвига sll, **srl**, **sla**, **sra**, **rol**, **ror**. Левым операндом должен быть вектор битов или булевских переменных, а правым – целое.

Операторы сложения + | - | &. Операнды для сложения и вычитания должны быть численного типа. Оператор конкатенации определен для произвольного одномерного регулярного типа.

Операторы знака + | - определены для любого численного типа.

Операторы произведения * | / | **mod** | **rem** определены для целых чисел. Операторы умножения и деления еще определены для типа с плавающей запятой. Но не допускается, чтобы оба операнда были физического типа, поскольку не определен тип результата.

Другие операторы abs | ** определены для любых численных типов, но при возведении в степень правый операнд должен быть целого типа.

Преобразование типов

Для того чтобы разнотипные объекты участвовали в выражениях без конфликтов, необходимо путем преобразований типов привести эти объекты к одному типу – типу выражения. Если типы близкие, то можно выполнить переход типа. Такими типами считаются типы real и integer, регулярные типы с одинаковым числом элементов того же самого типа и с одинаковыми диапазонами индексов. Переход от real к integer выполняется как:

```
B:=integer(10.5);
```

и результатом будет округление до единиц, т.е. 11.

Если типы не близкие, то необходимо выполнить вызов функции преобразования типа, многие из которых входят в состав стандартных пакетов библиотеки IEEE. Для подтипов, у которых один родоначальный тип, например, для natural и positive, преобразование типа не требуется.

1.4 Процессы и последовательные операторы

Основным элементом поведенческого описания устройства является такой параллельный оператор, как процесс. Тело процесса образуют последовательные операторы, к которым относятся: присваивание переменной, операторы **if**, **case**, **loop**, **next**, **exit**, **assert**, **report**, **wait**, **return**, **null**, вызов процедуры и присваивание сигналу.

Оператор процесса

Оператор процесса описывается в соответствии со следующим упрощенным синтаксисом:

```
\оператор процесса\ ::= process [(\список чувствительности)] [is]
                        {\объявление в процессе\}
                    begin
                        {\последовательный оператор\}
                    end process;
```

Процесс представляет собой неявно заданный бесконечный цикл. Симулятор исполняет последовательные операторы в теле процесса в заданном порядке, и после исполнения последнего оператора возвращает управление к первому оператору. После инициализации процесса в начале моделирования процесс оказывается или в активном состоянии, или приостановленным для ожидания заданных событий.

Процесс приостанавливается после исполнения оператора **wait**. Если в заголовке процесса есть список чувствительности, то внутри его операторы **wait** не допускаются, и в этом случае сам компилятор вставляет оператор **wait** в конце цепочки последовательных операторов. Процесс продолжает свое выполнение после оператора **wait**, если произойдет указанное в нем событие или изменится какой-либо сигнал в списке чувствительности.

Оператор if

Этот условный оператор в зависимости от заданных условий выполняет цепочки последовательных операторов, причем от условия зависит, которая из цепочек операторов выполняется. Упрощенный синтаксис оператора:

```
\оператор if\::=if \условие 1\ then
        {\последовательный оператор 1\}
[ { elsif  \условие 2\ then
        {\последовательный оператор 2\}]
   [else
{\последовательный оператор 3\}]
    end if;
```

Каждое из условий должно быть выражением, вычисляющим результат булевского типа. При выполнении этого оператора условия проверяются последовательно друг за другом, пока результат условия не будет **true**. Тогда выполняется соответствующая этому условию цепочка операторов и исполнение данного оператора **if** прекращается.

Оператор case

Этот оператор разрешает выполнение одной из цепочек последовательных операторов в зависимости от значения выражения селектора. Его синтаксис:

```
\оператор case\::=case \простое выражение\ is
     when \альтернативы\ => {\последовательный оператор\}
    {when \альтернативы\ => {\последовательный оператор\}}
    end case;
\альтернативы\:= \альтернатива\{ | \альтернатива\}
```

В выражении селектора \простое выражение\ должен стоять объект целого, перечисляемого или регулярного типа. Каждая из альтернатив \альтернатива\ должна быть такого же типа, что и \простое выражение\ и представлена выражением или диапазоном, например, 0 **to** 4 (в случае, если селектор – перечисляемого типа). Никакие два значения, получаемые из выражений альтернатив, не должны быть равны друг другу, т.е. множества альтернатив не перекрываются. Последней альтернативой может быть ключевое слово **others**, которое указывает на не перечисленные Альтернативы. Если слово **others** не применяется, то в альтернативах должны быть перечислены все возможные значения селектора.

Вот так можно запрограммировать приоритетный шифратор для четырехразрядного операнда-селектора DI с помощью этого оператора:

```
case DI is
  when "0000"=>DO<="000";
  when "0001"=>DO<="001";
  when "0010"|"0011"=>DO<="010";
  when "0100"|"0101"|"0110"|"0111"=>DO<="011";
  when others=> DO<="100";
end case;
```

Оператор цикла

Этот оператор несколько раз выполняет последовательность операторов. Его упрощенный синтаксис:

```
\оператор цикла\::=[\схема итерации\]loop
        {\последовательный оператор\}
    {next  [when \условие\];}
    {exit  [when \условие\];}
end loop;
\схема итерации\::=while \условие\
                        | for \переменная цикла \ in \диапазон\
```

По первой схеме итераций цикл, ограниченный ключевыми словами **loop** и **end loop** будет выполняться, пока условие \условие\ не примет значение false. Причем, это условие проверяется до выполнения цикла и если оно равно false, то цикл не выполняется. В примере:

```
    i:=1;  or_vec:='0';
while i<=n loop
        or_vec:= or_vec or vec(i);
         i:=i+1;
    end loop;
```

вычисляется переменная or_vec, равная функции ИЛИ от всех разрядов n – разрядного вектора vec. Если n = 0, то цикл не вычисляется. Этот пример можно записать с помощью второй схемы итерации как:

```
    or_vec:='0';
for i in 1 to n loop
        or_vec:= or_vec or vec(i);
    end loop;
```

Здесь переменная цикла i последовательно принимает значения 1,2,... из диапазона 1 **to** n. Переменную цикла не нужно объявлять, как другие переменные и ей нельзя выполнять присваивания.

Если необходимо завершить очередную итерацию до ее окончания, то применяют оператор **next** запуска следующей итерации. В примере

```
    numb:=0;
for i in 1 to n loop
   next when vec(i)='0';
   numb:=numb+1;
    end loop;
```

вычисляется число единиц в векторе vec.

Если нужно закончить оператор цикла до завершения всех итераций применяют оператор **exit** выхода из цикла. В примере

```
        numb:=7;
    for i in 7 downto 1 loop
       exit when d(i)='1';
       numb:=numb-1;
    end loop;
```

благодаря оператору **exit**, находится номер самой левой единицы в 7-разрядном векторе d, т.е. реализована функция приоритетного шифратора (сравните с примером оператора **case**).

Оператор **loop** иногда применяется без схемы итерации, т.е. когда цикл может выполняться неопределенно большое число раз.

Операторы assert и report.

Эти операторы были введены в язык VHDL для выявления ошибок моделирования и сообщения о них на консоль. У оператора ловушки **assert** следующий синтаксис:

```
\оператор assert\::= assert \булевское выражение\
            [report \строка сообщения\][severity \выражение\];
```

Здесь \булевское выражение\ – проверка какого–либо условия правильно-

сти моделирования, которое равно false, если найдена ошибка и true, если моделирование верно; \строка сообщения\ - выражение типа string, представляющее строку сообщения о причине ошибки. Выражение \выражение\ имеет предопределенный тип severity_level, состоящий из элементов note, warning, error и failure. Оно означает уровень критичности ошибки и при самом высоком уровне failure моделирование останавливается.

1.5 Подпрограммы

К числу подпрограмм относятся процедуры и функции. Подпрограмму задают с помощью описания спецификации подпрограммы. Она описывает действия при вызове подпрограммы и имеет следующий упрощенный синтаксис:

```
\спецификация подпрограммы\::=procedure \имя процедуры\[(\список параметров\)]
    | function \имя функции\[(\список параметров\)] return \тип результата\
                    is
        {\объявление в подпрограме\}
      begin
        {\последовательный оператор\}
      end [procedure][\имя процедуры\]

\список параметров\::=(\элемент списка\ {; \элемент списка\})
\элемент списка\::=[constant | variable | signal ]
        \идентификатор\{,\идентификатор\}: [in | out | inout] \тип параметра\
            [ := \статическое выражение\]
```

Здесь \имя процедуры\, \имя функции\ - идентификатор процедуры или функции. В списке параметров указывается информация о формальных параметрах подпрограммы. Вместо формальных параметров подставляются фактические параметры во время вызова подпрограммы. В каждом из элементов списка параметров может объявляться, какой это параметр (константа, переменная или сигнал) направление передачи параметра (**in**, **out** или **inout**) его тип или подтип и его значение по умолчанию.

Список параметров может отсутствовать, т.е. он тогда задан неявно. При этом имена фактических параметров совпадают с именами формальных параметров, объявленных в подпрограмме.

Ниже показан пример описания процедуры сортировки двух чисел.

```
procedure SORT2(variable x1,x2:inout integer) is
    variable t:integer;  -- объявление в подпрограме
begin
    if x1>x2 then       -- последовательный оператор
      return;           -- немедленное прекращение вызова процедуры
    else
      t:=x1; x1:=x2; x2:=t;
    end if;
end procedure;
```

Вызов подпрограммы содержит имя подпрограммы и список фактических параметров для передачи значений в подпрограмму и из нее. Связы-

вание формальных и фактических параметров может быть позиционным, поименованным или комбинированным. Вызов процедуры с позиционным связыванием:

```
SORT2(a1,a2);
```

Тот же вызов с ассоциативным связыванием:

```
SORT2(x2=>a2,x1=>a1);
```

При вызове подпрограмм типы фактических и формальных параметров должны строго совпадать. Если фактический параметр – константа, то она может быть задана выражением.

В вызванной подпрограмме операторы выполняются последовательно, пока не выполнится последний оператор или не попадется оператор **return**. Вызов функции всегда должен оканчиваться оператором **return** с выражением, вычисляющим возвращаемую величину.

Функции

Вызов функции используется как часть выражения, и он возвращает одиночное значение. Допускаются функции, которые имеют параметр in (считается по умолчанию), относящиеся к сигналам и константам (константа – по умолчанию). Обычно вызов функции не дает побочного эффекта. Это означает, что никакой объект, объявленный снаружи функции, не может быть изменен при ее вызове и для одних и тех же аргументов функция возвращает один и тот же результат.

Кроме идентификатора, обозначением функции может быть символ операции. Так, функция сложения векторов может быть объявлена как:

```
function "+"(a,b:bit_vector) return bit_vector;
```

Вызов такой функции не отличается от использования оператора сложения. Не допускается применение в функции оператора **wait**. Поэтому функция всегда выполняется моментально и возвращает значение сразу после своего вызова.

Процедуры

Используя параметры в режиме **out** или **inout**, процедуры могут возвращать произвольное количество значений. Вызов процедуры – это отдельный оператор, который бывает как последовательным, так и параллельным. Если не указывать режим параметра процедуры, то подразумевается, что он – режим **in**. Параметры могут относиться к классам констант, переменных, сигналов или файлов. Если класс параметра не указан, то входной параметр – константа, а выходной параметр – переменная.

Процедуры могут использовать оператор **wait,** и поэтому параллельный вызов процедуры с **wait** ведет себя как оператор процесса.

Перегрузка подпрограмм

В языке допускается, чтобы различные подпрограммы имели одинаковое имя. Тогда говорят, что такая процедура или функция перегружает другую процедуру или функцию. При компиляции и реализации вызова подпрограммы из набора подпрограмм с одинаковым именем выбирается подходящая подпрограмма, в которой соответствуют число, порядок, имена (если поименованное связывание параметров), и типы параметров.

1.6. Параллельные операторы

Как указывалось в разделе 1.2, описательную часть архитектуры составляют параллельные операторы. Такие операторы исполняются паралЛельно и поэтому их порядок в программе безразличен. Оператор вставки компонента уже был описан в разделе 1.2. Он реализует основной механизм задания структуры проекта. Поведенческое описание проекта задается другим параллельным оператором – процесссом, описанным в разделе 1.4. Также параллельными операторами являются параллельное присваивание сигналу, параллельный вызов процедуры и параллельный оператор **assert**.

Параллельное присваивание сигналу

Присваивание сигналу, вставленное в программу вне оператора процесса или подпрограммы, считается параллельным оператором. Такой оператор эквивалентен процессу с этим присваиванием, после которого стоит оператор **wait**. Следующие два оператора эквивалентны.

```
S<=A+B+1;
process(A,B) begin
    S<=A+B+1;
end process;
```

Логические функции можно задавать разновидностями этого оператора, такими как условное присваивание сигналу и селективное присваивание сигналу.

Условное присваивание сигналу

Оно означает выбор одного из выражений для присваивания в зависимости от условия и имеет упрощенный синтаксис:

```
\условное параллельное присваивание\::= \имя\<=
                                {\выражение\ when \булевское выражение\ else}
                                    \выражение\[when \булевское выражение\];
```

Этот оператор описывает эквивалентный процесс с оператором **if**. Поведение трехвходового мультиплексора описывается таким условным присваиванием:

```
Y<=A when SEL="00"
     B when SEL="01" else C;
```

Селективное присваивание сигналу

Такое присваивание эквивалентно процессу с оператором **case**. Оно имеет следующий упрощенный синтаксис:

```
\селективное параллельное присваивание\::= with \выражение\ select
                    {\имя\<= {\выражение\ when \альтернативы\,}
                                \выражение\[when others ];
```

где \альтернативы\ имеют то же значение, что и в операторе **case**. Пример оператора **case** в разделе 1.4 может быть представлен следующим селективным присваиванием:

```
with DI select
    DO<="000" when "0000",
          "001" when "0001",
          "010" when "0010"|"0011",
          "011" when "0100"|"0101"|"0110"|"0111",
          "100" when others;
```

Параллельный вызов процедуры

Этот оператор эквивалентен процессу, который содержит только эту процедуру, за которой стоит оператор **wait**. Такая процедура активизируется, как только возникнет изменение какого-либо параметра – сигнала с режимом **in** или **inout**. Процедура, используемая в параллельном вызове, не должна иметь параметров класса переменных.

Оператор generate.

Если необходимо неоднократно повторить один или несколько параллельных операторов, то используют оператор **generate**. Его упрощенный синтаксис:

```
\оператор generate\::= \метка\: for \идентификатор\ in \диапазон\ generate
                        [{\объявление\}
                    begin]
                        { \параллельный оператор\}
                    end generate [\метка\];
```

Метка оператора **generate** необходима для обозначения сгенерированной структуры, \идентификатор\ - это параметр оператора **generate**, а фраза \диапазон\ - диапазон его изменения. Они имеют такие же синтаксис и семантику, как и в операторе **loop**.

Для того чтобы управлять структурой проектируемого устройства, используется условный оператор **generate**. Его упрощенный синтаксис:

```
\условный оператор generate\::= \метка\: if \булевское выражение\ generate
                    [  {\объявление\}
                begin]
                    { \параллельный оператор\}
                end generate [\метка\];
```

В зависимости от условия, заданного булевским выражением, оператор вставляет или нет структуру устройства узлы, представленные параллельными операторами. В примере:

```
ANDn: if \подключить_AND2\=1 generate
      ANDi:  for i in 1 to n generate
                U_ AND2: AND2(A(i),B(i),T(i));
       end generate;
    end generate;
```

если целое значение \подключить_AND2\ равно 1, то выполняется вставка n компонентов AND2, к входам которых подключены разряды шин A и B.

1.7 Атрибуты

В языке VHDL сигналы, переменные и другие объекты, кроме своего значения, также имеют множество атрибутов. У каждого типа объектов есть несколько предопределенных атрибутов. Пользователь также может ввести ряд специальных атрибутов. Атрибуты бывают различного типа: атрибут – тип, значение, сигнал, функция, диапазон.

Атрибут объекта записывается как:

\имя объекта\' \имя атрибута\ .

Ниже рассматриваются некоторые предопределенные атрибуты.

Атрибуты для скалярного типа.

T'left – самое левое значение множества элементов скалярного типа T.
T'right – самое правое значение множества элементов скалярного типа T.
T'image(X) – функция строкового представление выражения X типа T.
T'value(X) – функция значения типа T от строкового представления X.
T'pos(X) – функция номера позиции элемента X в множестве типа T.
T'val(X) –функция значения элемента типа T стоящего в позиции X.
Примеры атрибутов:

```
    type st is (one,two,three);
st'right = three,   st'pos(three) = 2, st'val(1) = two.
positive'left =  1, positive'right =2147483647.
integer'value("1_000") =1000,  integer'image(330) ="330".
```

Атрибуты для регулярного типа.

A'left – левое значение диапазона индексов.
A'right - правое значение диапазона индексов.
A'range – диапазон индексов.
A'reverse_range – обратный диапазон индексов.
A'length – протяженность диапазона индексов.

Например, если заранее неизвестна длина вектора vec, то в примере, описанном в разделе 1.4. функцию ИЛИ можно вычислить так:

```
    or_vec:='0';
for i in vec'range loop -- или так:  for i in vec'left to vec'right loop …
      or_vec:= or_vec or vec(i);
end loop;
```

Атрибуты сигналов

S'event – сигнал, равный true, если произошло событие в сигнале S в данном цикле моделирования.

S'last_value – сигнал такого же типа, что и S, содержащий значение S до последнего события в нем.

Примером применения атрибутов сигналов является следующий процесс, моделирующий синхронный триггер.

```
process(CLK) begin
   if CLK='1' and CLK'event then-- D-триггер
      q1<=a1;
    end if;
end process;
```

1.8 Пакеты для проектирования вычислительных устройств

Стандартная технология проектирования вычислительных устройств основана на применении библиотеки IEEE. Эта библиотека состоит из более десятка пакетов и предназначена, в основном, для дискретного моделирования микросхем на всех уровнях, кроме уровня транзисторов, а также для синтеза логических схем. Далее будут рассмотрены особенности пакетов numeric_bit и std_logic_1164, которые используются в лабораторных работах.

Пакет numeric_bit

Этот пакет предназначен для стандартизации и упрощения программирования и моделирования устройств, в которых выполняются арифметические операции, а также для обеспечения последующего их синтеза. В пакете numeric_bit определены типы unsigned и signed как векторы элементов типа bit, а также ряд арифметических функций, функций сравнения и сдвига над операндами этих типов, которые перегружают одноименные функции и операции, определенные над целыми числами.

Значения типов bit_vector, unsigned и signed могут быть поразрядно равны между собой. Но для операнда unsigned перегружается такая функция, которая обрабатывает этот операнд как код числа без знака. Т.е. тип unsigned обозначает положительные двоичные целые числа. Аналогично тип signed соответствует двоичным целым числам со знаком в дополнительном коде.

В пакете определены перегружаемые логические функции **and**, **or**, **not**, **nand**, **nor**, **xor**, **xnor**, арифметические функции '+', '-', '*', **abs**, **mod**, **rem**, функции сдвига **srl**, **sll**, **ror**, **rol**, а также все функции сравнения для операндов типа unsigned, signed и integer в различном их сочетании. Причем векторы операндов могут иметь различные диапазоны. Например, если объявлены:

```
constant A:signed(4 downto 0) :="11100";  --число -4
constant B:unsigned(3 downto 1):="101";   --число 5,
```

то результаты различных функций будут следующие: A+7="00011", A+B = "00001", A-B = "10111", B*3 = "01111", (A = B) = false, (A<B) = true.

Функция Resize добавляет к операнду типа unsigned слева столько нулей, чтобы в результирующем векторе было общее число разрядов, равное второму операнду типа integer. Аналогично к операнду типа signed она добавляет слева разряды, равные старшему – знаковому разряду операнда. Если параметр длины меньше, чем длина вектора, то соответствующее число старших разрядов отбрасывается. Например:

Resize(unsigned("10"),5) = "00010", Resize(signed("10"),5) = "11110".

Функции взаимного преобразования типов To_Integer, To_Unsigned и To_Signed преобразуют аргументы типа integer, unsigned, signed, в результаты типа integer, unsigned и signed, соответственно. Например, вышеприведенные константы A и B могут быть объявлены как:

```
constant A:signed(4 downto 0) := To_Signed(-4,5) ;  --число -4
constant B:unsigned(3 downto 1):= To_Unsigned(5,3); --число 5
```

Так как типы signed, unsigned и bit_vector для векторов одинаковой длины - это тесно связанные типы, то преобразование типов таких векторов выполняется как переход типа. Та же константа B может быть объявлена как:

```
constant B:unsigned(3 downto 1):= unsigned(1-A(2 downto 0));   --число 5
```

Для удобства моделирования синхронных схем памяти определены булевские функции Rising_Edge и Falling_Edge определения фронта и спада сигнала, соответственно. Например, операторы

```
C<=Rising_Edge(CLK);
C<=CLK='1' and CLK'event;
```

выполняют присваивания одинакового значения булевскому сигналу C.

Пакет numeric_std имеет такие же функции, что и пакет numeric_bit, но они перегружают их для аргументов типа std_logic и std_logic_vector (см. ниже).

Пакет std_logic_1164

В пакете определены основные типы языка, а также логические функции и функции преобразования типов. Базовый тип std_ulogic представляет собой перечисляемый тип, который состоит из 9 элементов. Его объявление представлено в разделе 1.3. Базовый тип std_ulogic_vector является одномерным регулярным типом из элементов std_ulogic.

Подтип std_logic состоит из элементов типа std_ulogic, над которыми определена функция разрешения. Функция разрешения Resolved разрешает конфликты, когда два источника сигнала типа std_logic соединены выходами. Логика функции разрешения построена на приоритетах элементов типа

std_ulogic_vector, а также на обозначении конфликтных ситуаций элементами сильное неизвестное 'X' и слабое неизвестное 'W'. Так, не инициализированное 'U' имеет максимальный приоритет. Затем идет 'X' . '0' и '1' имеют меньший приоритет, чем 'X', но больший, чем 'W' . Слабые ноль и единица 'L' и 'H' имеют меньший приоритет, чем 'W', но больший, чем 'Z'. Если два источника выдают '0' и '1', то результат - 'X', а если 'L' и 'H' – то результат 'W'.

Таким образом, элементы 'U', 'X', 'W' и '-' кодируют не логические состояния а состояния сигнала в процессе моделирования, которые отличаются от правильного состояния. По такому состоянию можно определить, что произошла ошибка в функционировании модели, степень ее важности и возможно, причину ошибки.

Элементы '0', '1', 'Z', 'L' и 'H' не только кодируют состояния операндов булевской логики, но и позволяют моделировать схемы с общей шиной, реализованной как тристабильная шина или шина, реализующая логику "монтажного И" или "монтажного ИЛИ". Например, следующие операторы моделируют шину, к которой подключены источники – схемы с открытым коллектором и нагрузочный резистор:

```
C<='H';                          -- модель нагрузочного резистора
C<='0' when EA='0' else 'Z'; -- модель первого источника
C<='0' when EB='0' else 'Z'; -- модель второго источника
```

У сигнала C есть три источника и согласно функции разрешения, C принимает значение 'H' при условии, когда и EA, и EB имеют значение не '0', а иначе C принимает значение '0', т.е. это функция "монтажное И". Причем при любой правильной комбинации входных сигналов сигнал C не принимает значения 'Z', т.к. значение 'H' – "сильнее" его.

Следует отметить, что данная модель будет корректной, только если сигнал C объявлен как std_logic. Если он объявлен как std_ulogic, то компилятор выдаст сообщение об ошибке, так как над этим типом не определена функция разрешения. Как раз этим тип std_ulogic и подтип std_logic различаются друг от друга.

В пакете также определены перегружаемые логические функции, такие как **and**, **or**, **not**, **nand**, **nor**, **xor**, **xnor**. Логика этих функций в отношении к элементам типа std_ulogic аналогична логике функции разрешения, т.е. при ошибочных сигналах – операндах результат будет 'U' или 'X'. Кроме того, как и в реальном вентиле, слабые уровни 'L' и 'H' усиливаются до '0' и '1', а слабая ошибка 'W' усиливается до 'X'. Если на одном входе такое значение, которое само определяет результат, то на другом входе может быть произвольное значение. Например:

'X' **or** '1' = '1' , 'U' **and** '0' = '0', '-' **nand** '0' = '1'.

Для преобразования типов, объявленных в пакете в типы bit и bit_vector и обратно определены функции преобразования типов:

```
function To_Bit  ( s : std_ulogic; xmap : bit := '0') return  bit;
function To_Bitvector(s:std_logic_vector; xmap: bit := '0') return bit_vector;
function To_StdULogic( b : bit  ) return std_ulogic;
function To_StdLogicVector( b : bit_vector) return std_logic_vector;
```

и другие. В них операнд xmap представляет собой элемент, которым необходимо заменять элементы 'U', 'X', 'Z', 'W', '-' при преобразовании. Например, если объявлено

```
constant EL:std_logic_vector(0 to 8):="01HLZWXU-";
```

то To_bitvector(EL,'1') = "0110111111", To_bitvector(EL) = "011000000".

Также определены перегружаемые функции Rising_Edge и Falling_Edge для параметров типа std_ulogic. Эти функции возвращают значение true, если в текущем цикле моделирования их сигнал-аргумент имел фронт или спад, соответственно.

Вызов функции Rising_Edge(clk) не намного короче эквивалентной ему фразе: **if** clk = '1' **and** clk'event, но позволяет устранить некоторые неточности моделирования сигналов типа std_logic. Так, например, если моделируется триггер со срабатыванием по фронту сигнала, то он сработает правильно для перепада сигнала: 0 – H, если применяется функция Rising_Edge и не сработает, если использована вышеприведенная фраза.

2. ЭЛЕМЕНТНАЯ БАЗА МИКРОСХЕМ С ПРОГРАММИРУЕМОЙ ЛОГИКОЙ

2.1 Общая характеристика

Программируемые логические матрицы (ПЛМ) появились три десятилетия назад на базе постоянного запоминающего устройства (ПЗУ). ПЛМ представляет собой матрицу логических элементов с триггерами, в которых программируются конституенты единиц дизъюнктивных нормальных форм функций этих элементов. В первых ПЛМ программирование выполнялось пережиганием перемычек между источниками переменных и входами логических элементов. Затем перемычки заменили МОП-транзисторами с плавающим затвором, как в электрически перепрограммируемом ПЗУ, т.е. теперь ПЛМ изготовляются по технологии флэш-памяти. Большие ПЛМ (CPLD) отличаются только тем, что несколько ПЛМ собраны на одном кристалле и объединены программируемым полем связей.

Программируемые логические интегральные схемы (ПЛИС) появились полтора десятилетия назад как альтернатива ПЛМ. ПЛИС представляет собой матрицу маловходовых (от двух до пяти входов) логических элементов, триггеров, отрезков линий связи, соединяемых перемычками из полевых транзисторов. Судя по английскому названию - Field Programmable Gate Array (FPGA) - ПЛИС программируются изменением уровня электрического поля (field) в затворах этих транзисторов. Затворы всех "программмирующих" полевых транзисторов подключены к выходам триггеров сдвигового регистра, который заполняется при программировании ПЛИС. Некоторые из участков этого регистра могут также играют роль ячеек ПЗУ.

Прошивка обычно хранится в ПЗУ, стоящем рядом с ПЛИС. После включения питания или по сигналу сброса она переписывается в программмирующий сдвиговый регистр ПЛИС. Этот процесс называется конфигурированием. Так как основу ПЛИС составляют триггеры, хранящие прошивку, то ПЛИС изготавливаются по технологии микросхем статического ОЗУ.

2.2. Логический элемент ПЛМ

Один элемент ПЛМ 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: $M \leq 5$, $N \leq 52$, $K \leq 52$. Для увеличения M используются PLM –логические расширители, что эквивалентно последовательному включению PLM.

Например, допустимо такое задание логической функции на языке VHDL:

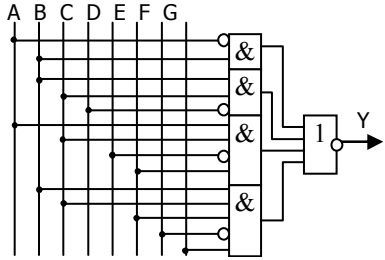


Рис.2.1. Функциональная схема, соответствующая оператору VHDL

```
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);
```

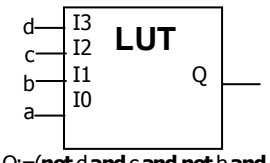
для которой $M = 4$, $N = 5$, $K = 9$. Здесь A, B, C, D, E, F, G, L – входные булевские или битовые переменные, а Y – выходной сигнал – результат. Такой оператор присваивания сигналу соответствует функциональной схеме на рис.2.1.

2.3. Логический элемент ПЛИС

В ПЛИС программируемая логическая таблица (LUT– lookup table) используется как базовый логический элемент. Программируемая логическая таблица 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, 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.2.2 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.2.2. Логическая таблица и ее содержание

Любое логическое выражение на VHDL транслируется компилятором - синтезатором в схему из одной или нескольких LUT с соответствующим содержимым. Логические схемы можно реализовать в проекте путем вставки компонента. Библиотека Unisim содержит модели всех компонент ПЛИС фирмы Xilinx. Например, компонент LUT из этой библиотеки можно задать таким интерфейсом:

```
component LUT4 is generic(
```

```
INIT: BIT_VECTOR);
  port(O          :  out  STD_ULOGIC;
    I0            :  in   STD_ULOGIC;
    I1            :  in   STD_ULOGIC;
    I2            :  in   STD_ULOGIC;
    I3            :  in   STD_ULOGIC);
end component;
```

где 16- разрядный вектор INIT задает содержимое таблицы, кодирующее требуемую логическую функцию. В той же библиотеке можно выбрать компоненты, реализующие любую логическую функцию от четырех переменных, которая в результате отображается в LUT.

Кроме логических таблиц, ПЛИС имеют другие программируемые ресурсы для эффективного построения арифметических и логических схем. Это схемы-мультиплексоры распространения переноса, схемы поразрядного произведения, схемы для формирования многовходовых мультиплексоров. В современных ПЛИС также имеются отдельные блоки комбинациионных умножителей. Для использования этих ресурсов необходимо иметь библиотеку с их описанием и пользоваться операторами вставки компонентов.

Современные компиляторы-синтезаторы достаточно умело распознают ситуации, когда оператор или выражение соответствует тому или иному варианту сочетания логических ресурсов ПЛИС. Так, операция "+" в VHDL – описании при компиляции заменяется схемой сумматора, а операция "*" - схемой умножителя. Но во многих случаях результирующая синтезироВанная схема может быть неоптимальной.

Во-первых, компилятор-синтезатор строит схему, например, сумматор-вычитатель, в соответствии с алгоритмом построении таких схем, ориентированном на общий случай. И тогда такая схема содержит избыточные узлы. Так, выражение A+B+1 будет реализовано на двух сумматорах, а не на одном, как при ручном проектировании. Во-вторых, синтезированная схема после размещения и разводки, как правило, имеет неоптимальное быстродействие из-за больших задержек в длинных маршрутах межсоединений. Это объясняется тем, что, размещение и разводка никогда не выполняются с учетом регулярности структуры устройства и сводится к отысканию локального, а не глобального оптимума.

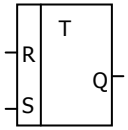
Чтобы получить устройство с минимальными аппаратными затратами и максимальным быстродействием приходится его описывать на уровне LUT или ПЛМ. Затем путем управления размещением и разводкой с помощью файла ограничений или атрибутов вставляемых LUT или ПЛМ можно добиться такого размещения, чтобы удовлетворялся принцип близкодействия, обеспечивающий минимум задержек межсоединений. Поэтому, чтобы научиться получать оптимизированные проекты, некоторые из лабораторных работ основаны на структурном описании блоков на базе LUT или ПЛМ.

2.4. Триггеры в ПЛМ и ПЛИС

При выполнении лабораторных работ предлагается реализовать проектируемые блоки на базе CPLD или ПЛИС. В первом случае базовый элемент – это элемент ПЛМ с триггером (PLMT), а во втором – LUT и D - триггеры. Рассмотрим программирование D – триггеров.

Триггер представляет собой элемент с памятью. Т.е. его выходной сигнал зависит от состояния триггера, запомненного в предыдущие моменты времени. Такое поведение нельзя описать параллельным оператором присваивания. В большинстве случаев поведение триггера описывается в операторе процесса. Например, в следующем процессе описывается RS-триггер.

```
process(R,S) begin
if R='1' and S='0'  then
   Q<='1' after td;
elsif  R='0' and S='1' then
   Q<='0' after td;
elsif  R='1' and S='1' then
  report "некорректные данные RS -триггера";
end if;
end process;
```

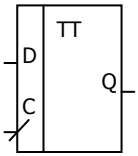


При запуске этого процесса по изменению сигнала R или S если R='1', то триггер устанавливается в 1, а если S='1', то триггер устанавливается в 0. При других запусках процесса триггер не изменяет своего состояния, т.е. его поведение соответствует поведению RS-триггера. Причем если R='1' и S='1', то симулятором выдается сообщение о некорректной работе триггера.

Приведенный в этом примере триггер относится к числу асинхронных триггеров. Такие триггеры сейчас очень редко используются при проектировании микросхем с их описанием на VHDL, т.к. в схемах с их применением очень трудно достичь ожидаемого уровня работоспособности и повторяемости результатов проектирования. Поэтому в проектах, как правило, применяются синхронные триггеры и регистры на их основе.

Синхронный триггер меняет свое состояние на новое только по фронту или спаду синхросигнала, приходящего на его синхровход. В вычислительной технике наибольшее распространение получили D - триггеры. Такой триггер описывается следующим процессом.

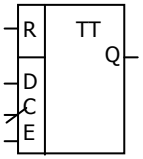
```
process(C) begin
        if C='1' and C'event then
                Q<=D;
        end if;
end process;
```



В этом процессе запись в триггер Q происходит в случае события перехода сигнала C из 0 в 1, т.е. по фронту этого сигнала. В логическом выражении условия использован атрибут сигнала C'event, который равен true, если в момент запуска процесса произошло изменение этого сигнала.

Триггеры, применяемые в ПЛИС и CPLD, имеют также вход разрешения записи E, вход R асинхронной установки в ноль (вход сброса) или вход S асинхронной установки в единицу. Такой триггер с асинхронным сбросом описывается в следующем процессе.

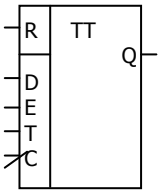
```
process(C,R) begin
  if R='1' then
      Q<='0';
 elsif C='1' and C'event then
    if E='1' then
       Q<=D;
    end if;
  end if;
end process;
```



Следует отметить, что аналогичным образом описывают и регистры. Так, если в приведенных примерах сигнал Q – это вектор из *n* бит, то такой процесс описывает *n* - разрядный регистр.

Для построения счетчиков иногда применяются T-триггеры, которые можно описать следующим процессом.

```
process(C,R) begin
  if R='1' then
      Q<='o';
 elsif C='1' and C'event then
     if E='1' then
         Q<= D;
     elsif T='1' then
         Q<= not Q;
     end if;
  end if;
end process;
```



В этом триггере по сигналу асинхронного сброса R триггер устанавливается в ноль, по сигналу E выполняется синхронная начальная установка триггера с входа D, а по сигналу T - инвертируется его состояние.

Часто триггер вводится в проект с помощью оператора вставки компонента. Следующий оператор вставляет компонент FDRE-триггера с асинхронным сбросом и разрешением записи.

```
FF: FDRE port map (Q=>Q, D=>D, C=>CLK, CE=>E, R=>R);
```

В этом операторе использовано поименованное связывание имен портов триггера с входными и выходным сигналами. Оригинальная модель этого компонента описана в библиотеке Unisim.

2.5.Блоки памяти в ПЛИС

При выполнении лабораторных работ предлагается реализовать RAM на базе ПЛИС фирмы Xilinx. В случае, когда объем памяти не превосходит 4096 бит, следует использовать модуль памяти RAM16X1 объемом 16 бит, описанный в библиотеке Unisim. Интерфейс этого модуля выглядит следующим образом

```
component RAM16X1 is
port ( D : in std_ulogic;   -- входное данное
   WE : in std_ulogic;      -- разрешение записи
   WCLK: in std_ulogic;     -- синхросигнал для записи
   A0 : in std_ulogic;      -- биты адреса
   A1 : in std_ulogic;
   A2 : in std_ulogic;
   A3 : in std_ulogic;
   O  : out std_ulogic); -- выходное данное
end component;
```

Запись в модуль происходит по спаду сигнала WE, а чтение выполняется постоянно, если WE=0. Порты модуля имеют тип std_ulogic. Модуль памяти на 256 шестнадцатиразрядных слов имеет следующий интерфейс.

```
component RAMB4_S16 is
   port (DI: in STD_LOGIC_VECTOR (15 downto 0);--входное данное
     EN  : in STD_ULOGIC;  -- разрешение обращения
     WE  : in STD_ULOGIC;  -- сигнал записи
     RST : in STD_ULOGIC;  -- асинхронный сброс
     CLK : in STD_ULOGIC;  -- сигнал синхросерии
     ADDR: in STD_LOGIC_VECTOR (7 downto 0);--адрес
     DO  : out STD_LOGIC_VECTOR (15 downto 0) );--выходное данное
end component;
```

Аналогичный интерфейс имеют модуль RAMB4_S1 на 4096 однобитных данных, модуль RAMB4_S2 2048 двухразрядных слов, RAMB4_S4 - 1024 четырехразрядных слов и RAMB4_S8 - 512 восьмиразрядных слов. При этом шины адреса и данных имеют соответствующую разрядность. Запись в модуль происходит по фронту синхросигнала CLK при WE=1, а чтение – также по фронту синхросигнала, если WE=0.

Постоянное запоминающее устройство (ROM) реализуется как блок RAM с заданным начальным состоянием и без использования функции записи. При этом начальное состояние записывается в этот блок из файла конфигурации при конфигурировании ПЛИС. Ниже приведен пример описания блока ROM, который можно использовать как ROM программ.

```
    library IEEE;
use IEEE.std_logic_1164.all;
--pragma translate_off
library UNISIM;              --библиотека с моделями ОЗУ, реализуемыми в ПЛИС
--pragma translate_on
entity ROM is  port( CLK : in BIT;
    RST :    in BIT;     -- сброс блока ОЗУ
```

```
    ENA:     in BIT;    -- разрешение работы
    ADDRI : in BIT_VECTOR(15 downto 0);  --  адрес
    INSTR : out BIT_VECTOR(15 downto 0););--считанная команда
end ROM;
architecture ROM_SYNT of ROM is
  component ramb4_s16      -- Block_RAM объемом 256 16-разрядных слов
    --pragma translate_off
    generic(   INIT_00, INIT_01: BIT_VECTOR); --начальное состояние Block_RAM
    --pragma translate_on
    port(
      DI : in std_logic_VECTOR(15 downto 0);
      EN : in std_ulogic;
      WE : in std_ulogic;
      RST : in std_ulogic;
      CLK : in std_ulogic;
      ADDR : in std_logic_VECTOR(7 downto 0);
      DO : out std_logic_VECTOR(15 downto 0));
  end component;
  attribute INIT_00:  string; -- ячейки памяти с 0 по 15
  attribute INIT_01:  string; -- ячейки памяти с 16 по 31
--начальное состояние Block_RAM , программируемое в ПЛИС
  attribute INIT_00 of U_PROM: label is "00000000000000000000000e4418104" &
                                        "00000000000000000000000e3318104" ;
  attribute INIT_01 of U_PROM: label is "0000000000000000000350391fcb402" &
                                        "330132107551744173316 20f610c7001";
   constant gnd:std_ulogic:='0';
   signal clki,enai,rsti:std_ulogic;
  signal addri,di,doi:std_logic_VECTOR(15 downto 0);
begin
   clki<= To_StdULogic(CLK);
   rsti<= To_StdULogic(RST);
   enai<= To_StdULogic(ENA);
   di<=(others=>gnd);
   addri<=To_StdLogicVECTOR(ADDR);
U_PROM:  ramb4_s16
  --pragma translate_off     --состояние памяти модели Block_RAM
  generic map(INIT_00 => X"0000_0000_0000_0000_0000_0000_e441_8104" &
                         X"0000_0000_0000_0000_0000_0000_e331_8104" ,
              INIT_01 => X"0000_0000_0000_0000_0000_3503_91fc_b402" &
                         X"3301_3210_7551_7441_7331_620f_610c_7001")
  --pragma translate_on
  port map(
    DI =>di,
    EN =>ENA,
    WE=>gnd,
    RST =>rsti,
    CLK =>clki,
    ADDR =>addri(7 downto 0),
    DO =>doi);
     INSTR<= To_BITVECTOR(doi);
```

```
end ROM_SYNT;
```

В этой архитектуре начальное состояние ROM задается дважды. Первый раз оно задается в настроечных константах **generic**, что необходимо для передачи состояния в модель BlockRAM, которая подключается из библиотеки Unisim. Второй раз оно задается в атрибутах пользователя attribute INIT, которые указывают компилятору-синтезатору, какое состояние BlockRAM должен иметь при конфигурировании. Так как компонент BlockRAM, вставляемый при синтезе, не должен иметь настроечных констант, то в тексте программы этот интерфейс и связывание настроечных констант окружены прагмами *--pragma translate_off* и *--pragma translate_on* для исключения из компиляции.

Настроечная константа и соответствующий атрибут кодируют 256 последовательных битов BlockRAM, начиная со старшего бита. Таким образом, одна настроечная константа представляет собой 16 соседних ячеек памяти, причем первые 16 бит справа кодируют младшую ячейку. В настроечной константе отдельные ячейки разделены подчеркиванием. Для того чтобы строка не была слишком длинной, настроечная константа и строка атрибута представлены как конкатенации битовых векторов и строк, соответственно.

Современные компиляторы-синтезаторы распознают операторы процесса, которые описывают поведение RAM и в результирующую схему подставляют соответствующий блок RAM из библиотеки компонентов или строят схему, которая содержит необходимое число таких блоков. Например, следующий процесс распознается как RAM и исполняется на базе компонента RAMB4_S16.

```
  type MEMO256x16 is array (0 to 255) of BIT_VECTOR(15 downto 0);
  signal RAM: MEMO256x16:=(others=>"0000000000000000");
  signal ADDR: BIT_VECTOR(7 downto 0); -- 8-разрядный адрес
  signal DI,DO: BIT_VECTOR(15 downto 0); -- входное и выходное данные
  signal WE: BIT;                        --разрешение записи
. . .
RAM_4K:process(CLK,WE,ADDR)
 variable address:natural;
 begin
   address:=  To_Integer(Unsigned(ADDR));  --перевод адреса в целое число
    if Rising_edge(CLK) then
      if WE='1' then
        RAM(address)<= DI;                  --запись входного данного
      end if;
    end if;
    DO<= RAM(address);                      --чтение по адресу ADDR
 end process;
```

2.6. Тристабильные схемы в ПЛИС и CPLD

Действие двунаправленных шин, часто применяемых в компьютерной схемотехнике, основано на том, что несколько источников шины электрически подключены к одной линии через тристабильные буферы. Корректность функционирования шины достигается за счет того, что одновременно к шине подключается с логическими уровнями 0 или 1 не более чем один буфер, а остальные буферы включены в третье – высокоомное состояние, обозначенное как 'Z'. При этом правильно разработанная логическая схема управления буферами гарантирует, что два буфера не будут открыты одновременно.

В ПЛИС и CPLD каждый выход, называемый pad, может быть запрограммирован как двунаправленный или тристабильный. В ПЛИС фирмы Xilinx применяется тристабильный буфер, который имеет следующий интерфейс.

```
component BUFT is
port( O: out   STD_ULOGIC; -- выход
      I: in    STD_ULOGIC;      -- вход
      T: in    STD_ULOGIC);     -- управляющий вход, если=1, то
end component;                  -- на выходе – 'Z'
```

Если несколько сигналов имеют тип, производный от std_logic, то выходы источников этих сигналов можно объединять вместе, так как над ними определена функция разрешения конфликтов. На этом свойстве основано программирование логики с тремя состояниями. Например, четырехвходовой мультиплексор, основанный на двунаправленной шине x, можно запрограммировать так:

```
    x <= a when (s = "00") else (others=>'Z');
x <= b when (s = "01") else (others=>'Z');
x <= c when (s = "10") else  (others=>'Z');
x <= d when (s = "11") else  (others=>'Z');
```

В этой модели источники сигналов x, порождаемые операторами параллельного присваивания, отображаются в буферы с тремя состояниями, которые при истинном соответствующем условии выбора выдают сигналы a,b,c,d типа std_logic или высокоимпедансное состояние 'Z' на все разряды шины x.

Современная элементная база ПЛИС не содержит внутренних буферов с тремя состояниями. Поэтому если двунаправленная шина запрограммирована с применением тристабильной логики, то компилятор-синтезатор построит эквивалентную двунаправленную шину, основанную на комбинационных схемах мультиплексоров.

2.7 Пакет CNetwork

Для удобства проведения лабораторных работ и моделирования устройств была разработана библиотека функций и элементов, описанная на

VHDL. Библиотека хранится в файле CNetwork_Lib.VHD (компьютерная схемотехника – Computer Network engineering).

В библиотеке хранится пакет функций CNetwork и модели LUT, LSM, триггеров, а также модель генератора случайных чисел. 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 CNetwork 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 CNework;
```

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. 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".

Правда, эти функции при синтезе приводят к чрезмерным аппаратным затратам. Поэтому для оптимизированного синтеза рекомендуется применять эквивалентные функции из пакета IEEE.Numeric_BIT.

Функция LOG_TAB(e,d,c,b,a,mask) возвращает значение логической функции, реализованную в виде таблицы. Ее содержимое задается вектором битов mask, например, LOG_TAB(**open**,'0','0','0','1',"0111111111111111"). В этом случае она возвращает '0', если на адресных входах все нули, 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. На основе этой функции строятся модели LUT и эти модели корректно распознаются при синтезе.

Пакеты и объекты VHDL-проекта группируются в одну библиотеку после своей компиляции. Приведенный в библиотеке вариант модели элемента PLM имеет следующее объявление объекта.

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

Один из алгоритмов функционирования объекта PLM описан в следующем примере его архитектуры, обозначаемой как PLM(STENCIL).

```
    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;
```

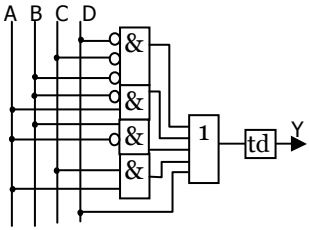


Рис. 2.3. Функциональная схема элемента PLM

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. 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.2.3.

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

```
use CNetwork.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 CNetwork.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;
```

Описание **use** указывает компилятору, что для компиляции объекта необходимо использовать пакет CNetwork, а ключевое слово **all** – что в этом пакете могут быть использованы все его составляющие элементы. Файл CNetwork_Lib.VHD с этим пакетом находится в рабочем каталоге и он должен быть 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 левый бит в векторе (с номером 15) заносится в младшую ячейку таблицы. Например, функция ИЛИ от четырех переменных кодируется ими 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 CNetwork.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. Из этой же 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 выходных данных. Каждое данное стробируется сигналом CLK. Необходимо указывать начальное состояние SEED, для того, чтобы несколько генераторов не выдавали одинаковые последовательности данных.

Файлы пакета даны в приложении.

3. ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ПРОЕКТИРОВАНИЯ

3.1. Система Active HDL

Запуск системы

VHDL-проект размещается в системе Active-HDL в отдельном каталоге в рабочей области (Project space). Система Active-HDL запускается при активизации криптограммы (нажатием кнопки)  . При этом система запрашивает, открыть ли существующую рабочую область (Open existing workspace) или создать новую рабочую область (Create new workspace). Если нужно создать новую рабочую область, то система приглашает указать имя и каталог этой области. Затем можно создать пустую область (Create an empty design) или создать каталог проекта и поместить в него существующие файлы проекта (Add existing resource files). В последнем случае нужно выбрать добавление файлов (Add files), найти нужный каталог, отметить курсором в нем один файл или группу соседних файлов (держать нажатой кнопку Shift), или несколько файлов (держать нажатой кнопку Ctrl).

Если требуется совместная работа с компилятором-синтезатором и САПР ПЛИС, управляемыми из системы, то их указывают в окнах Synthesis tool и Implementation tool, соответственно. При этом указывают тип микросхемы в окне Default Family. В конце установки рабочей области указывают имя проекта и его каталог размещения (если необходимо).

Управление системой

Система Active HDL управляется с помощью манипулятора "мышь". При этом курсор устанавливают на строку меню, пиктограмму, объект проекта, которые активизируют нажатием левой кнопки мыши. При нажатии правой кнопки мыши всплывает дополнительное меню команд и опций. Многие команды управления системой связаны с "горячими" клавишами.

Стандартным способом управления Active HDL, как и любым другим симулятором, является задание управляющих команд в окне консоли. Такое управление обычно автоматизируют путем объединения управляющих команд в макрофайл - DO–файл. Запуск DO – файла – это стандартный способ автоматизации процесса моделирования. Он часто применяется для тестирования проектов на всех стадиях разработки и поставляется заказчику вместе с проектом.

Графический интерфейс системы

Главная панель системы Active HDL показана на рис.3.1. Она включает в себя область меню, окно обхода проекта (Browser), окно консоли, рабочее

окно и многие другие, которые устанавливаются в области рабочего окна в том или ином режиме работы системы.

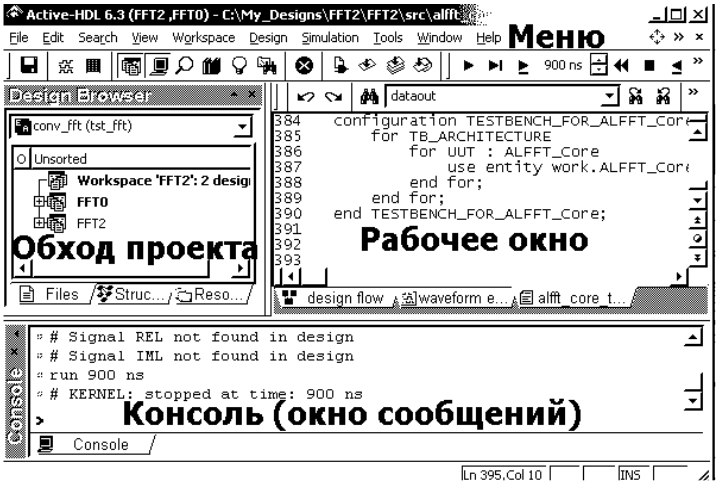


Рис.3.1. Главная панель системы Active HDL

Область меню включает в себя систему всплывающих меню и группы пиктограмм, связанных с наиболее часто применяемыми режимами работы системы. В том числе оно включает в себя пиктограммы управления запуском и остановкой моделирования, окно задания периода моделирования и окно, указывающее текущий момент моделирования. В режиме настройки (Customize) можно установить желательный набор пиктограмм и окон.

Окно обхода проекта имеет окно дерева файлов, библиотек проекта и управления файлами (Files), окно дерева компонентов проекта, его сигналов, констант и переменных (Structure), окно вспомогательных файлов проекта (Resources) и окно выбора вершины проекта (строка вверху).

В окне консоли выдаются сообщения о ходе и результатах компиляции, моделирования, найденных ошибках и т.п. В нем задаются также строковые управляющие команды для симулятора.

В рабочем окне выставляются рабочие окна проекта. Это могут быть окна текстовых редакторов для программ на VHDL, Verilog, C, DO–файлов, графические редакторы структурных схем проекта (Block diagram), графа алгоритма автомата (State diagram), условных графических обозначений компонентов проекта (Symbol editor), которые используются для ввода и редактирования объектов проекта. Для выдачи результатов моделирования используются окно графиков (Waveform editor), окно протокола изменения сигналов (List) и другие. Для управления проектом при взаимодействии с внешними компиляторами-синтезаторами и САПР ПЛИС используют окно хода проектирования (Design flow manager). Одновременно могут быть

открыты несколько рабочих окон, снабженных внизу флажками. Требуемое окно выбирается по флажку.

Добавление и редактирование файлов

Для добавления VHDL-файла к проекту выбирают курсором его место в дереве файлов в окне обхода проекта (Add new file), нажатием правой кнопки открывают меню добавления файлов и выбирают или создание файла (New – VHDL-source), или начинают поиск готового файла в файловой системе (Add files to design).

При вводе и редактировании VHDL-файла используют те же возможности, которые предоставляются обычными текстовыми редакторами. Библиотека шаблонов (Language assistant), открываемая кнопкой  , предоставляет возможность выбрать типичные шаблоны операторов и конструкций VHDL и вставить их в текст.

Для подключения к проекту стандартных библиотек и выбора из них интерфейсов компонентов, чтобы их вставить в свой текст, используют управление библиотеками (Library manager), открываемое кнопкой  .

Компиляция файлов

После ввода и редактирования VHDL-файла его компилируют. Компиляция вызывается кнопкой  . При этом ошибки, найденные при компиляции выдаются в окне консоли. Одновременно строка с ошибкой подчеркивается в тексте красной волнистой линией.

При групповой компиляции всех файлов проекта используют кнопки  и  . В первом случае файлы компилируются подряд, а во втором – с изменением порядка, чтобы объекты проекта, которые являются компонентами, компилировались раньше, чем объекты, в которых они вставлены. Скомпилированные VHDL - файлы, не имеющие ошибок и замечаний, подключаются к результирующему проекту и отмечаются птичкой на дереве файлов в окне управления файлами. Когда все файлы проекта успешно скомпилированы, задают корневой объект проекта в окне выбора вершины проекта. После этого возможно моделирование проекта.

Задание графиков сигналов

При моделировании проекта его поведение обычно наблюдают в окне графиков, которое вызывают с помощью кнопки  . В этом окне задают графики сигналов, которые требуется наблюдать при моделировании. Для этого можно использовать кнопку  добавления сигналов. Обычно эти графики задают "перетаскиванием" сигналов из окна обхода дерева компонентов проекта и его сигналов.

Для каждого графика можно задать режим отображения – Properties, установку которого вызывают нажатием клавиш Alt+Enter или из меню, вызванного правой кнопкой мыши. При установке основных свойств (General) можно указать основание системы счисления отображаемых векторов битов или целых чисел. При установке свойств самого графика

(Display) задают высоту графика в пикселах (Heigh), его цвет и форму: Literal – указание значения, Logic – логические уровни и Analog – график кривой. В последнем случае задают диапазон отображаемых чисел, которые соответствуют уровням усечения сигнала при имитации аналогового усиления с насыщением.

При моделировании выбранным сигналам можно принудительно задать постоянное или изменяющееся во времени значение, например, подать сигналы на входные порты объекта. Режим программирования стимуляторов вызывают активизацией кнопки  . При этом генератором сигнала может быть генератор прямоугольных колебаний (Clock), временная последовательность различных уровней (Formula), постоянное значение (Value), нажатие клавиши (Hotkey), генератор линейно изменяющегося сигнала (Counter), предварительно заданный в редакторе график сигнала (Custom), предварительно определенный и запомненный график (Predefined).

Используя меню обращения с файлами, окно с графиками сигналов можно сохранить в файле с расширением AWF или восстановить, прочитав выбранный AWF–файл. Отмеченные графики можно «перетащить» в другие приложения, например, в редактор текста, используя операции «поместить в карман», «выложить из кармана» (клавиши Ctrl-C, Ctrl-V).

Моделирование проекта

Перед запуском моделирования устанавливают временной интервал моделирования. Например, интервал 1000 нс устанавливается так: 

. Если корневая архитектура или конфигурация проекта задана, то ее моделирование запускается кнопкой  . При этом в окне текущего времени вместо сообщения: No simulation наблюдается изменение времени в наносекундах или пикосекундах, в зависимости от минимального интервала изменения сигнала в модели. Кроме того, в этом же окне со знаком + указывается текущий номер дельта-цикла моделирования.

В процессе моделирования кнопка остановки моделирования имеет красный цвет:  . Моделирование всегда можно остановить, нажав на эту кнопку.

Через заданный интервал времени моделирование останавливается. При этом в окне графиков вырисовываются графики заданных сигналов. Моделирование можно продолжить, нажав кнопку  или привести в исходное состояние, нажав на кнопку  .

После каждой компиляции любого файла проекта состояние симулятора необходимо привести в исходное состояние, нажав на кнопку , так как измененный проект может быть запущен на моделирование только сначала.

3.2. Система проектирования ПЛИС

В цикле лабораторных работ рекомендуется применять систему проектирования ПЛИС Xilinx ISE Webpack. Эту систему можно получить через Интернет с сайта фирмы Xilinx. Ее возможностей достаточно как для изучения технологии подготовки проектов для ПЛИС и сложных ПЛМ фирмы Xilinx, так и для получения файлов конфигурации для современных ПЛИС с объемом до сотен тысяч вентилей. Эта система представляет собой программную оболочку и библиотеку программ для автоматизированного ввода проектов синтеза логических схем, их моделирования, размещения и разводки в пространстве ПЛИС, генерации файлов прошивки, анализа временных задержек, энергопотребления и т.п., а также соответствующего визуального отображения результатов различных этапов проектирования.

Для вызова системы Webpack используется пиктограмма  . Программная оболочка (Project Navigator, рис.3.2) включает в себя всплывающие меню: File – операции с файлами проекта, Edit – редактирования файлов, View – настройки системы окон, Project – команды управления множеством файлов проекта, Source – операции с файлами – источниками проекта, Process – запуск исполнения выбранного этапа проектирования, Window – управление используемыми окнами, Help – подсказки и справочная система. Также она включает в себя ряд пиктограмм, дублирующих команды всплывающего меню и набор открытых окон.

В окне Sources in project (файлы проекта) отображено дерево файлов, включенных в проект как файлы исходных данных. Это VHDL–файлы .VHD, EDIF-файлы .EDN, файлы ограничений .UCF и другие. В окне Processes for source xxx (процессы проектирования) отображено дерево процессов проектирования и соответствующих файлов результатов для головного файла проекта xxx, выбранного в окне Sources in project. В рабочем окне отображается и редактируется содержимое выбранных текстовых файлов. В окне сообщений выводятся сообщения программ, выполняемых в процессе проектирования.

Для создания проекта выполняют команду New project. В ряде предложенных окон-шаблонов задают имя проекта, каталог размещения файлов проекта и тип вершины проекта: HDL – файл на VHDL или Verilog, EDIF – файл описания логической схемы в формате EDIF, NGC/NGO - файл описания логической схемы, синтезированной в Xilinx Webpack или Schematic – схемный файл. Затем выбирают элементную базу проекта, тип файла вершины проекта (HDL), программу для синтеза (XST), язык сгенерированной модели после синтеза (VHDL).

Далее добавляют к папке проекта новый исходный файл, нажимая на кнопку New source и задавая имя нового файла и его тип (VHDL-module – файл проекта, VHDL-package – файл пакета и т.п.) и указывая его место в каталоге файлов. После этого вводят новый файл проекта с помощью редактора. Если ввод нового файла не требуется, то к проекту добавляют

существующие исходные файлы при нажатии кнопки Add source. При этом пользуются предлагаемым деревом файлов.

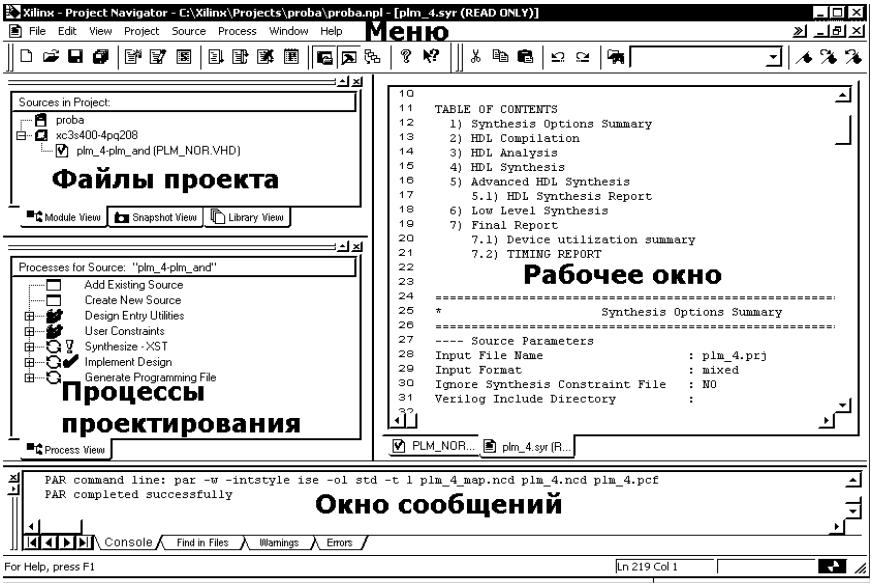


Рис.3.2. Панель Xilinx Webpack

Для запуска автоматического проектирования выбирают курсором корневой файл проекта в окне файлов и курсором инициализируют требуемый этап проектирования в окне процессов проектирования. Например, для синтеза логической схемы выбирают в дереве процессов процесс Synthesize - XST, для получения проекта в ПЛИС – процесс Implement Design, а для генерации файла прошивки и дальнейшего конфигурирования ПЛИС - процесс Generate Programming File. В окне сообщений отображается ход проектирования. И если при этом обнаружены ошибки в проекте, то в этом же окне отображаются диагностические сообщения о них.

После выполнения процесса проектирования можно ознакомиться с результатами проектирования, открыв соответствующие файлы отчетов (Report–файлы), выбрав их в окне процессов. В файле Synthesis Report (.SYR) важными сведениями являются отчет об аппаратурных затратах - Device utilization summary и отчет о задержках в проекте - Timing Summary. Также в этих файлах можно прочитать сообщения об ошибках, причины их возникновения и типичные способы их устранения.

В файле Place & Route Report (.PAR) находится отчет об аппаратурных затратах - Device utilization summary и отчет о задержках в проекте - Delay Summary Report. Интегральными показателями качества проекта являются суммарный объем аппаратурных затрат - Number of Slices в эквивалентных

конфигурируемых блоках (CLB slices) и минимальная длительность тактового интервала - TS_CLK в наносекундах.

Для изучения того, как компилятор-синтезатор распознал функциональную схему, описанную на уровне регистровых передач, запускают программу View RTL Schematic, которая представляет функциональную схему в графическом виде. При этом, пользуясь командами Pop и Push, выполняют переходы между уровнями иерархии многоуровневого чертежа проекта.

Управлять ходом проектирования можно путем установки настроек процессов проектирования, с помощью редактирования файла ограничений проекта и атрибутов пользователя в VHDL-файлах. Для установки настроек процесса проектирования выбирают в дереве процессов требуемый процесс, нажимают правую кнопку мыши, выбирают в меню команду Properties и редактируют таблицу настроек процесса.

Файл ограничений (.UCF–файл) для выбранного корневого файла проекта генерируется после вызова одной из программ генератора такого файла. После вызова Create Timing Constraints генерируются временные ограничения проекта, самым важным из которых является минимальный период синхросигнала. После вызова Assign Package Pins генерируются установки привязки портов проекта к конкретным выводам микросхемы ПЛИС. Также файл ограничений может быть отредактирован вручную после его вызова с помощью Edit Constraints.

Для генерации файлов моделей после размещения и после разводки кристалла выбирают процессы Generate Post-Map Simulation Model и Generate Post - Place&Route Simulation Model. Сгенерированные VHDL–файлы представляют собой модели исходного устройства, уточненные до задержек в вентилях - в первом случае и до задержек в ЭКЛБ, триггерах, линиях связи – во втором.

Подсистема Xilinx CORE Generator (COREGen) встроена в программную оболочку ISE и предназначена для генерации параметризованных вычислительных модулей, оптимизированных для реализации в ПЛИС фирмы Xilinx. В отличие от модулей, спроектированных пользователем, COREGen-модули, как правило, имеют большее быстродействие и меньшие аппаратные затраты, так как они представлены в виде оптимизированной схемы на уровне логических таблиц и триггеров, к которым привязана информация об их взаимном размещении в ПЛИС.

Подсистемой COREGen генерируются модули в формате EDIF с расширением EDN, а также файл шаблона компонента этого модуля на языке VHDL с расширением VHO. Этот файл используется как текст, который вставляется в VHDL–файл в операторе вставки компонента сгенерированного модуля, таким образом, чтобы, во-первых, модуль был правильно встроен в проект и, во-вторых, проект с таким модулем мог быть смоделирован.

4. ЛАБОРАТОРНЫЕ РАБОТЫ

4.1. Общие требования к лабораторным работам

Выполнение лабораторных работ состоит в проектировании вычислительных блоков с верификацией их функционирования. Каждый такой блок описывается на языке VHDL, а его поведенческая верификация выполняется на симуляторе. Для проверки того, что блок описан синтезируемым стилем и для определения реальных аппаратных затрат и временных параметров блока его схему синтезируют на уровне вентилей с помощью системы Webpack. Таким образом, выполнение лабораторной работы соответствует ходу проектирования, применяемому в промышленности. Для контроля знаний, полученных при выполнении лабораторной работы, предлагается ответить на вопросы, приведенные в конце ее описания.

Перед выполнением лабораторной работы в соответствии с номером варианта, который зависит от номера зачетной книжки, в [2] на с.115-117 выбирается задание на выполнение лабораторной работы. Параметры задания включают: тип логического элемента (PLM или LUT), максимальное число термов PLM или количества входов LUT, разрядность результи-рующей схемы, перечень операций, перечень выходных сигналов и т.п.

Обачно выполнение лабораторной работы имеет три стадии: разработка поведенческой модели блока, разработка структурной модели блока и разработка испытательного стенда с проверкой функционирования блока.

Разработка поведенческой модели блока

Поведенческая модель блока описывается стилем потоков данных, если блок представляет собой комбинационную схему или поведенческим стилем, если в состав блока входят регистры, триггеры и другие элементы памяти. При этом можно использовать операций с целыми числами и функции из пакетов CNetwork или IEEE.Numeric_BIT. Для описания блока используется редактор и компилятор VHDL, входящие в состав пакета Active HDL.

После того, как блок описан в виде архитектуры, он может быть протестирован путем ручной подачи входных тестовых значений при ее моделировании.

Разработка структурной модели блока.

Структурная модель блока описывается структурным стилем. При этом используются компоненты из файла CNetwork.VHD. Для этого используется редактор и компилятор VHDL, входящие в состав пакета Active HDL. Поведенческое и структурное описания имеют одинаковое объявление объекта, но различные архитектуры. В первом случае архитектура имеет имя, например, BEH а во втором - STR.

Разработка испытательного стенда и тестирование моделей.

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 моделью. В лабораторных работах обычно в качестве эталонной модели используется разработанная поведенческая модель блока, а в качестве тестируемой модели – структурная модель. Тестируемые последовательности данных подается с генераторов случайных чисел. Объект такого генератора описан в файле CNetwork.VHD.

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.

Отчет по лабораторной работе.

Отчет по лабораторной работе должен содержать:

- сведения о студенте, включающие ФИО, группу, номер зачетной книжки,
- цель работы,
- описание заданного варианта блока,
- ход проектирования компонентов и структуру блока,
- тексты описаний поведенческой и структурной моделей блока, испытательного стенда,
- графики сигналов, снятых на испытательном стенде,
- измеренные задержки сигналов,
- выводы.

4.2. Арифметико-логическое устройство
(лабораторная работа 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 программмирования и отладки описания логических схем на языке VHDL.

Теоретические сведения

Устройство LSM предназначено для выполнения как арифметических (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 роли особого операнда выступает бит C_0 переноса в младший разряд. Кроме *n*-разрядного результата *Y*, результатами LSM часто выступают такие признаки, как перенос из старшего разряда C_n, переполнение *V*, признак нулевого результата *Z* и бит знака *S*. Тип операции LSM задается управляющим кодом *F*, кодировка которого выбирается в каждом конкретном случае особо.

Примеры описания LSM

Рассмотрим пример проектирования n=4 –разрядного LSM, который при *F*=00 выполняет операцию $A+B+C_0$, при F=01 – операцию $A-B-C_0$, при *F*=10 – операцию поразрядного И над *A* и *B* и при *F*=11 – операцию поразрядного ИЛИ, где C_0 – перенос в младший разряд. При этом кроме результата Y, выдается бит Z – признак нулевого результата и бит C_3 – перенос из старшего разряда.

Объявление объекта для такого LSM выглядит следующим образом.

```
use  CNetwork.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;
```

Поведенческая модель LSM.

В своем поведенческом описании объект представляется в виде алгоритма функционирования, не привязанном к элементной базе. Но, в отличие от алгоритма, заданного в виде обычной программы, здесь задается четкая последовательность операций. При этом операции выполняются последовательно-параллельно: последовательные операторы исполняются последовательно, а параллельные – параллельно. Здесь 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 входные 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 объявлены сигналы, которые используются как промежуточные сигналы в вычислениях. В начале описательной части выполняется преобразование типов сигналов – сигналов векторов бит A, B, C0 в сигналы ai,bi,ci целого типа с помощью функций преобразования из пакета CNetwork. Следует отметить, что в VHDL параллельные операторы можно отметить меткой, имя которой упрощает чтение и отладку программы. В операторе с меткой ADDER описывается сумматор-вычитатель, который в зависимости от условия F(0), сложение или вычитание целых чисел.

В операторе, 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. Результат сложения-вычитания преобразуется в 5- разрядный вектор с учетом разряда переполнения, а результаты логических операций дополняются нулем в старших разрядах. Это дополнение выполняется операцией конкатенации бит: '0'&(A **or** B). Благодаря этому, при присваивании вектору бит, разрядность вектора ybi, стоящего слева от оператора «<=» становится равной разрядности результата выражения справа от этого оператора.

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.

Этот пример архитектуры описан синтезируемым стилем. Но так как в нем не использованы стандартные функции преобразования типов, результат его компиляции будет иметь слишком высокие аппаратные затраты. Поэтому ниже представлен пример описания LSM с использованием стандартного пакета IEEE.Numeric_BIT, который компилируется в схему с малыми аппаратными затратами (всего 15 LUT).

```
architecture NUM of LSM is
      signal ai,bi,yi,bp1:SIGNED(4 downto 0);
      signal ybi:BIT_VECTOR(4 downto 0);
begin
      ai<= RESIZE(SIGNED(A),5);--добавили 1 знаковый разряд
      bi<= RESIZE(SIGNED(B),5);
      bp1<=bi    when C0='0' else  bi+1;      --прибавили перенос
      yi<= ai+bp1  when F(0)='0' else   ai-bp1;
MUX:with F select
      ybi<= BIT_VECTOR(yi) when "00"|"01",
            '0'&(A and B) when  "10",
            '0'&(A or  B) when others;
       C3<= ybi(4);
       Z  <='1' when ybi(3 downto 0)="0000" else '0';
       Y<= ybi(3 downto 0);
end NUM;
```

Структурная модель LSM на базе LUT.

Модель LSM, описанная структурным стилем, строится на основе элементов LUT. Такой проект взаимно однозначно отображается в прошивку ПЛИС, когда каждый из компонентов реализован в LUT микросхемы.

LSM имеет *n* ступеней. Каждую ступень можно представить объединением трех LUT, первая из них LNI , выполняет 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 LUT - 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.4.1.

Указанные LUT задаются таблицами истинности 4.1-4.3. По правой колонке таблиц 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.

Таблица 4.1. Таблица истинности LNI

Адрес	Действие	F_1,F_0	Bi	Ai	Xi	Адрес	Действие	F_1,F_0	Bi	Ai	Xi
0	A+B	00	0	0	0	8	A&B	10	0	0	0
1			0	1	1	9			0	1	0
2			1	0	1	10			1	0	0
3			1	1	0	11			1	1	1
4	A-B	01	0	0	1	12	A∨B	11	0	0	0
5			0	1	0	13			0	1	1
6			1	0	0	14			1	0	1
7			1	1	1	15			1	1	1

Таблица 4.2. Таблица истинности LNO

Адрес	Действие	F_1,F_0	Xi	Ci	Yi	Адрес	Действие	F_1,F_0	Xi	Ci	Yi
0	A+B	00	0	0	0	8	A&B	10	0	0	0
1			0	1	1	9			0	1	0
2			1	0	1	10			1	0	1
3			1	1	0	11			1	1	1
4	A-B	01	0	0	1	12	A∨B	11	0	0	0
5			0	1	0	13			0	1	0
6			1	0	0	14			1	0	1
7			1	1	1	15			1	1	1

Таблица 4.3. Таблица истинности LNC

Адрес	Действие	Ci	Bi	Ai	Ci+1	Адрес	Действие	Ci	Bi	Ai	Ci+1
		$F_0 = 0$						$F_0 = 1$			
0	A+B	0	0	0	0	8	A&B	0	0	0	0
1		0	0	1	0	9		0	0	1	1
2		0	1	0	0	10		0	1	0	0
3		0	1	1	1	11		0	1	1	0
4	A-B	1	0	0	0	12	A∨B	1	0	0	1
5		1	0	1	1	13		1	0	1	1
6		1	1	0	1	14		1	1	0	0
7		1	1	1	1	15		1	1	1	1

```
LNI:LUT4 generic map(mask=>X"E896")
              port map(a=>A(i),b=>B(i),c=> F(0),d =>F(1),Y =>X(i));
    LNO:LUT4 generic map(mask=>X"CC96")
              port map(a=>c(i),b=>X(i),c=> F(0),d =>F(1),Y =>yi(i));
    LNC:LUT4 generic map(mask=>X"B2E8")
             port map(a=>A(i),b=>B(i),c=> c(i),d =>F(0),Y =>c(i+1));
```

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.

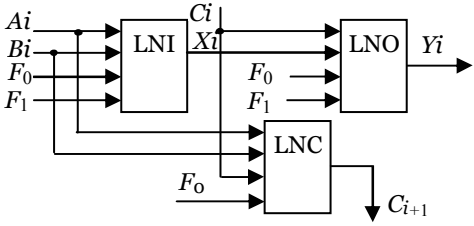


Рис.4.1. Структура одной ступени LSM

Для описания всего LSM можно продублировать эти операторы $n = 4$ раза. Но эффективнее это выполнить оператором размножения параллельных операторов **generate**. 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, b, c, 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"E896")
                  port map(a=>A(i),b=>B(i),c=> F(0),d =>F(1), Y =>X(i));
      LNO:LUT4 generic map(mask=>X"CC96")
                  port map(a=>C(i),b=>X(i),c=> F(0),d =>F(1), Y =>yi(i));
      LNC:LUT4 generic map(mask=>X"B2E8")
                  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;
```

Структура LSM, сгенерированная в системе Webpack, показана на рис.4.2. Ее аппаратные затраты составляют 13 LUT, т.е. синтезированное устройство имеет минимальные аппаратные затраты.

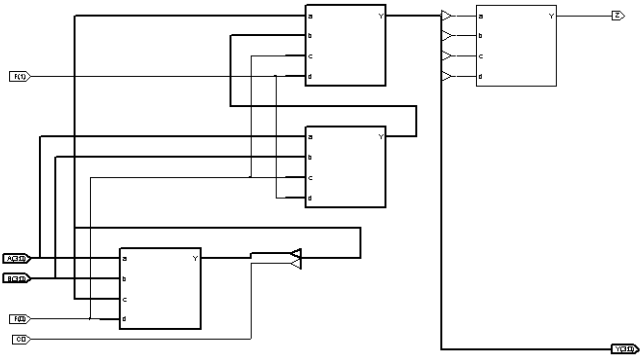


Рис.4.2. Структура LSM, спроектированного на базе LUT.

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  CNetwork.all;
entity PLM_4 is generic(td:time:=1 ns);        -- задержка
       port(a, b, c, d: in BIT;
                    Y : out BIT);
end PLM_4;
```

Для входной логической схемы LNI, согласно таблице истинности функции X_i (см. табл.4.1), архитектура выглядит следующим образом:

```
architecture PLMI of PLM_4 is
begin                -- арифметические операции ------------
      Y<=((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 описываются блоки LNO и LNS как архитектуры PLMO и PLMS, соответственно:

```
architecture PLMO of PLM_4 is
begin                -- арифметические операции
      Y<=((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                   -- арифметические операции
      Y<=((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:

```
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, b, c, 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(yi(3),yi(2),yi(1),yi(0),Z);  --Элемент ИЛИ-НЕ
      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 ошибку при связывании допустить трудно.

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 моделью. 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 эталонного 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=>CLK,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);
  COMP_Y:     Y<=Y1 xor Y2; --компараторы для сравнения результатов
  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 LSM(BEH) и LSM(STR_LUT). При вставке компонентов генераторов случайных чисел с помощью настроечных констант были заданы разрядности их выходов n=4 и разные начальные сосноояния, чтобы выдаваемые ими значения операндов A и B были различными.

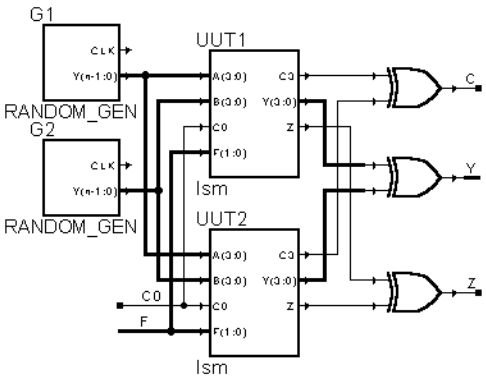


Рис.4.3. Структура испытательного стенда

Структура испытательного стенда, построенная с использованием утилиты Code2Graphics системы Active HDL, показана на рис. 4.3. 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 путем изменения их начального значения и перекомпиляции файла испытательного стенда.

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

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.

Разработка поведенческой модели LSM.

Поведенческая модель LSM описывается стилем потоков данных с использованием операций с целыми числами и функций из пакета CNetwork . 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.

Разработка структурной модели LSM.

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

Разработка испытательного стенда и тестирование моделей.

За образец испытательного стенда берется его пример, описанный выше. 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.

Вопросы по лабораторной работе.

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?
Что выполняет функция конкатенации в VHDL?

4.3. Счетчик команд

(Лабораторная работа 2)

Цель лабораторной работы:

овладеть знаниями и практическими навыками по проектированию таких комбинационных устройств с памятью, как счетчик команд (ICTR). Лабораторная работа также служит для овладения навыками программирования и отладки описания триггеров и логических схем на языке VHDL.

Теоретические сведения

Счетчик команд предназначен для вычисления текущего адреса команды в процессоре. В зависимости от вида текущей команды и признака условия, счетчик команд должен выдавать различные адреса новой команды. Если текущая команда – не команда перехода, или если признак условия не истинный, то адрес следующей команды равен текущему адресу плюс длина текущей команды до *k* байт. Если это команда перехода и условие истинно, то следующий адрес – адрес перехода, который, например, определяется на основе кода поля адреса текущей команды. И при начальной установке состояние ICTR должно быть установлено в ноль.

Таким образом, ICTR должен обладать функциями счетчика с инкрементом 1,2,...,*k*, с возможностями сброса и начальной установки. Для выполнения функций ICTR необходим малоразрядный сумматор с инкрементом SM, регистр RG младшей части адреса и счетчик – регистр CTR, хранящий старшие разряды адреса и считающий импульсы переполнения SM (рис.4.4).

Операции ICTR можно закодировать словом F, так как это выполнено в примере, показанном в табл. 4.4. Кодировка F выбирается в каждом конкретном случае особо. Целесообразно, чтобы часть кодовых комбинаций F совпадала с кодами приращений, как это указано в табл.4.4. Тогда упрощается схема управления ICTR.

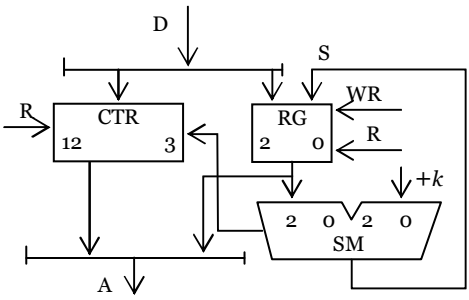


Рис. 4.4. Структура ICTR

В табл.4.4 указаны мнемоническое обозначение (МО) операции, значение Q^{t+1} равное состоянию *i*-го разряда сумматора, D_T означает тип операции, ICTR задается управляющим трехбитовым кодом *F*,

Примеры описания ICTR

Рассмотрим пример проектирования ICTR для параметра k=4 и числа входов логических схем

N=4, структура которого показана на рис.4.4, который функционирует согласно таблице 4.4.

Таблица 4.4 Кодирование операций ICTR

Описание операции	MO	F_2 F_1 F_0	Q^{t+1}	D_T	R
Без изменения	M	0 0 0	Q^t	S	0
Инкремент на 1	+1	0 0 1	S	S	0
То же на 2	+2	0 1 0	S	S	0
То же на 3	+3	0 1 1	S	S	0
То же на 4	+4	1 0 0	S	o	0
Сброс	R	1 0 1	S	X	1
Запись	WR	1 1 0	S	D	1
Безразлично	-	1 1 1	X	X	X

Объявление объекта для такого ICTR выглядит следующим образом.

```
library IEEE;
use IEEE.Numeric_BIT.all, use  CNetwork.all;
entity ICTR is
      port(CLK : in BIT;  -- синхровход
                R : in BIT;  -- сброс
               WR: in BIT;  -- сигнал записи
                D : in BIT_VECTOR(12 downto 0);  -- адрес перехода
                F : in BIT_VECTOR(2 downto 0);   -- функция
                A : out BIT_VECTOR(12 downto 0) );  -- выходной адрес
end ICTR;
```

Поведенческая модель ICTR.

Поведенческая модель объекта представляет собой алгоритм его функционирования, который может быть затем автоматически преобразован в соответствующую логическую схему с регистрами. Поведение ICTR зависит не только от входных данных, но и от состояния ICTR, изменяемого в предыдущие моменты времени. Поэтому устройство ICTR невозможно описать только операторами параллельного присваивания и необходимо использовать операторы процесса. Поведенческую модель ICTR можно описать в виде следующего описания архитектуры.

```
architecture BEH of ICTR is
signal RG: BIT_VECTOR(2 downto 0); -- регистр мл.части адреса
signal SM: unSIGNED(3 downto 0);  -- сумматор мл.части адреса
signal CTR:unSIGNED(12 downto 3);-- счетчик ст.части адреса
begin
-- описание сумматора ----------------------------------
SUM:SM<= Resize(unSIGNED(RG),4)+unSIGNED(F);
-- описание RG    ----------------------------------------
R_3:process(R,C)
begin
   if R='1' then
```

```
        RG<="000";
    elsif CLK='1' and CLK'event then
        case F is
            when "001"|"010"|"011"|"100"=>RG<=BIT_VECTOR(SM(2 downto 0));
            when "101" => RG<="000";
            when "110" => RG<=D(2 downto 0);
            when others=> null;
        end case;
    end if;
end process;

CT:process(CLK,R) -- описание счетчика ---------------------------------------
begin
    if R='1' then
        CTR<="0000000000";
    elsif CLK='1' and CLK'event then
        if F="101" then
            CTR<="0000000000";
        elsif F="110" then
            CTR<= D(12 downto 3);
        elsif (F(2) = '0' or F="100") and SM(3) ='1' then
            CTR<= unSIGNED(CTR)+1;
        end if;
    end if;
end process;
A<=BIT_VECTOR(CTR)&RG;  -- выходной адрес
end BEH;
```

В операторе, отмеченном меткой SUM, описан сумматор SM (см. рис.4.4). При этом векторы бит считаются числами без знака, на что указывает переход в подтип unSIGNED. 3 младшие разряда суммы SM(2 **downto** 0) подаются на вход регистра RG, а старший разряд SM(3)– как разряд переполнения - на счетный вход счетчика CTR.

В процессе R_3 описан трехразрядный регистр. В него по фронту синхросерии записывается ноль, 3 младших разряда входного данного D или сумма SM в зависимости от управляющего слова F, семантика которого указана в таблице 4.4. Логика управления регистром реализована оператором **case**.

В операторе процесса CT описано поведение 17- разрядного счетчика CTR. При этом состояние счетчика представлено битовым вектором, CTR, который рассматривается как число без знака. В первых пяти комбинациях слова F, если разряд переноса SM(3)=1, то к состоянию счетчика прибавляется 1. В зависимости от следующих двух комбинаций управляющего слова F состояние счетчика обнуляется или принимается новое значение с шины D. Результирующий адрес A формируется с помощью операции конкатенации из разрядов регистра RG и счетчика CTR.

Структурная модель ICTR на базе PLMT

Рассмотрим проектирование ICTR на базе PLMT. У всех PLM Одинаковое объявление объекта, такое, как приведено в лабораторной работе 1. Отличие лишь в количестве входов: PLM_4 – 4-входовая, а PLM_3 – 3-входовая PLM. Отдельные триггеры имеют интерфейс триггера FDRE, опиисанного в 2 главе.

Модель имеет структуру верхнего уровня, показанную на рис.4.4. В ней 3 узла будем проектировать раздельно как отдельные объекты. При этом учитываем, что поведение этих узлов соответствует трем операторам опиисанной выше архитектуры ICTR(BEH).

Проектирование SM

SM представляет собой 3-разрядный сумматор числа R с числом F в диапазоне от 0 до 4 с выходом переноса Q(3). Таблица 4.5 представляет собой таблицу истинности одного разряда сумматора.

Таблица 4.5 Логика 1 разряда сумматора

Ri Fi Qi	Qi+1 Si	Ri Fi Qi	Qi+1 Si	Ri Fi Qi	Qi+1 Si	Ri Fi Qi	Qi+1 Si
0 0 0	0 0	0 1 0	0 1	1 0 0	0 1	1 1 0	1 0
0 0 1	0 1	0 1 1	1 0	1 0 1	1 0	1 1 1	1 1

PLM, которые вычисляют i-й разряд переноса – PLM_Q и разряд суммы – PLM_S, приведены ниже при C = R_i, B=F_i, A=Q_i.

```
architecture PLM_Q of PLM_3 is    --вычисление Qi+1
begin
      Y<=((not C and B and A) or (C and not B and A) or ( C and B )
        )        after td;  --задержка элемента
end PLM_Q;
architecture PLM_S of PLM_3 is    --вычисление Si
begin
      Y<=((not C and not B and A) or (not C and B and not A)
        or ( C and not B and not A) or (C and B and A)
        )        after td;  --задержка элемента
end PLM_S;
```

Можно принять во внимание, что Q_0 =0, Q_1=R_0F_0 . Тогда первые разряды переноса и суммы можно вычислить в следующих PLM при D = R_1, C = F_1, B=R_0, A=F_0.

```
architecture PLM_Q1 of PLM_4 is  --вычисление Q1
begin
     Y<=((not D and C and B and A)or (D and not C and B and A) or ( D and C )
        )        after td;  --задержка элемента
end PLM_Q1;
architecture PLM_S1 of PLM_4 is   --вычисление S1
begin
     Y<=(( not D and not C and B and A) or (D and not C and not B)
        or (D and not C and not A) or (not D and C and not B)
```

```
        or (not D and C and not A) or (D and C and B and A)
        )       after td;  --задержка элемента
end PLM_S1;
```

Следует отметить, что все объекты PLM уже объявлены в файле CNetwork_lib.VHD, поэтому их объявлять в проекте второй раз не нужно. На основе этих PLM строим следующую структурную модель сумматора.

```
entity SM_3 is port(  --  3- разрядный сумматор
                    R:in BIT_VECTOR(2 downto 0);-- данное с регистра
                    F:in BIT_VECTOR(2 downto 0);-- инкремент
                    S:out BIT_VECTOR(2 downto 0);-- сумма
                    Q:out BIT);
end entity;

architecture PLM of SM_3 is
   constant gnd:BIT:='0';  -- нулевой бит
   signal qi2: BIT;        -- перенос в 2 разряд
begin

      S0:entity PLM_3(PLM_S) port map  -- 0 разряд суммы
            (a=>gnd,b=>F(0),c=> R(0), Y =>S(0));
       S1:entity PLM_4(PLM_S1) port map -- 1 разряд суммы
            (a=>F(0),b=>R(0),c=> F(1),d=>R(1),Y =>S(1));
       S2:entity PLM_3(PLM_S) port map  -- 2 разряд суммы
            (a=>qi2,b=>F(2),c=> R(2),       Y =>S(2));
       Q2:entity PLM_4(PLM_Q1) port map  -- перенос в 2 разр.
            (a=>F(0),b=>R(0),c=> F(1),d=>R(1),Y =>qi2);
       Q3:entity PLM_3(PLM_Q) port map  -- выходной перенос
            (a=>qi2,b=>F(2),c=> R(2),Y =>Q);

end PLM;
```

Таким образом, вместо шести PLM в сумматоре применяется только пять. Как видим, здесь применена вставка архитектур одних и тех же объектов, а не вставка компонентов. Поэтому объявлять компоненты таких объектов в декларативной части архитектуры PLM не нужно.

Проектирование RG

Узел RG представляет собой регистр, на входе которого M подключен мультиплексор, пропускающий данное с выхода сумматора SM, входное данное D или выдающий ноль в зависимости от кода F. Если обусловить, что запись в регистр выполняется в каждом такте, то формирование сигнала разрешения записи в триггер не нужно. Тогда выход S сумматора подается на вход регистра при условиях: F=000|001|010|011|100, а вход D – при условии F=110, а в остальных случаях – ноль. Все разряды мультиплексора одинаковые, поэтому он реализуется на PLM одного типа при $E=S_i$, $D = D_i$, $C = F_2$, $B=F_1$, $A=F_0$:

```
architecture PLM_MX of PLM_5 is
begin
      Y<=((E and not C) or (E and C and not B and not A)-- пропуск S
        or (D and C and B and not A)     -- пропуск D
        )        after td;                    --задержка элемента
end PLM_MX;
```

Описание объекта регистра выглядит следующим образом.

```
entity RG_3 is port(CLK:in BIT;   --синхросигнал
                    R:in BIT;-- асинхронный сброс
                    F:in BIT_VECTOR(2 downto 0); -- управление
                    D:in BIT_VECTOR(2 downto 0); -- входное данное
                    S:in BIT_VECTOR(2 downto 0); -- сумма
                    RG:out BIT_VECTOR(2 downto 0)-- выход регистра
                    );
end entity;
architecture PLM of RG_3 is
component FDRE  is port (Q:out  BIT;--триггер, описан в CNetwork
    D, C, CE, R:in BIT);
  end component;
  constant one:BIT:='1';
  signal M:BIT_VECTOR(2 downto 0);-- выход мультиплексора
begin
MX_RG:for i in 0 to 2 generate
MUX: entity PLM_5(PLM_MX) port map  -- мультиплексор
     (E=>S(i), D=>D(i),C=>F(2),B=>F(1),A=>F(0),Y=>M(i));
REG: FDRE  port map    --регистр
  (C=>CLK,R=>R,CE=>one,D=>M(i),Q=>RG);
  end generate;
end PLM;
```

Проектирование CTR

Счетчик адреса можно представить в виде регистра и сумматора, прибавляющего к содержимому Q регистра сигнал переноса C_3 с сумматора SM. У этого сумматора должны быть функции генерации нуля при F = 101 для обнуления счетчика и пропуска входного данного D при F = 110, необходимого при переходе по новому адресу. Каждая схема формирования разряда сумматора D_T должна состоять из PLM формирования сигнала переноса C_i в следующий разряд и PLM сигнала возбуждения триггера. PLM переноса описывается в следующей архитектуре при $A = C_{i-1}$, $B=Q_i$.

```
architecture PLM_C of PLM_3 is   -- логика переноса
begin
      Y<=B and A after td;
end PLM_C;
```

PLM возбуждения триггера описывается в следующей архитектуре при $F=Q_i$, $E=C_i$, $D= D_i$, $C = F_2$, $B=F_1$, $A=F_0$:

```
architecture PLM_CT of PLM_6 is  -- выход сумматора счетчика
```

```
begin
   Y<=(((F and not E) or (not F and E)) and      --XOR
         (not C or (C and not B and not A)))--сумма
    or (D and C and B and not A) -- пропуск D
      after td;  --задержка элемента
end PLM_CT;
```

Тогда объект счетчика описывается следующим образом.

```
entity CTRG is port(CLK:in BIT;   --синхросигнал
                    R:in BIT;-- асинхронный сброс
                    C3:in BIT;-- перенос из сумматора
                    F:in BIT_VECTOR(2 downto 0);-- управление
                        D:in BIT_VECTOR(12 downto 3);-- входное данное
                        CTR:out BIT_VECTOR(12 downto 3)-- выход счетчика
                        );
end entity;
architecture PLM of CTRG is
component FDRE  is port (Q:out  BIT;
                          D, C, CE, R:in BIT);
   end component;
   constant one:BIT:='1';
   signal c:BIT_VECTOR(13 downto 3); -- переносы
   signal Q:BIT_VECTOR(12 downto 3); -- выход регистра
   signal Dt:BIT_VECTOR(12 downto 3);-- вход регистра
begin
   c(3)<=C3;
 CT:for i in 3 to 12 generate
     CARRY: entity PLM_3(PLM_C) port map  -- переносы
          (C=>one,B=>C(i),A=>Q(i),Y=>c(i+1));
     SM: entity PLM_6(PLM_CT) port map  -- сумматор
          (F=>Q(i),E=>C(i), D=>D(i),C=>F(2),B=>F(1),A=>F(0),Y=>Dt(i));
    REG: FDRE  port map     --регистр
          (C=>CLK,R=>R,CE=>one,D=>Dt(i),Q=>Q(i));
   end generate;
   CTR<=Q;       --выходной сигнал
end PLM;
```

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

```
architecture PLM of ICTR is
      component SM_3 is port(R:in BIT_VECTOR(2 downto 0);-- данное
                        F:in BIT_VECTOR(2 downto 0);-- инкремент
                             S:out BIT_VECTOR(2 downto 0);-- сумма
                             Q:out BIT);
      end component;
      component  RG_3 is port(CLK:in BIT;              --синхросигнал
                             R:in BIT;                  -- асинхронный сброс
                             F:in BIT_VECTOR(2 downto 0); -- управление
                             D:in BIT_VECTOR(2 downto 0);-- входное данное
```

```
                    S:in BIT_VECTOR(2 downto 0); -- сумма
                    RG:out BIT_VECTOR(2 downto 0) );-- выход регистра
    end component ;
    component  CTRG is port(CLK:in BIT;          --синхросигнал
                    R:in BIT;                    -- асинхронный сброс
                    C3:in BIT;                   -- перенос из сумматора
                    F:in BIT_VECTOR(2 downto 0);-- управление
                    D:in BIT_VECTOR(12 downto 3);-- входное данное
                    CTR:out BIT_VECTOR(12 downto 3) );-- выход счетчика
    end component;
    signal RG: BIT_VECTOR(2 downto 0);  -- регистр мл.части адреса
    signal SM: BIT_VECTOR(2 downto 0);  -- сумматор мл.части адреса
    signal CTR:BIT_VECTOR(12 downto 3);-- счетчик ст.части адреса
    Signal C3:BIT;                  --перенос
begin
---------- Сумматор ----------------------------
    U_SM:SM_3 port map(R=>RG,F=>F,-- инкремент
              S=>SM,                -- сумма
              Q=>C3);               -- перенос
---------- Регистр  ----------------------------------------
    U_RG:RG_3 port map(CLK=>CLK,R=>R,F=>F,-- управление
              D=>D(2 downto 0),     -- входное данное
              S=>SM,                -- сумма
              RG=>RG);              -- выход регистра
---------- Счетчик старших разрядов адреса ----------------
    U_CT:CTRG port map(CLK=>CLK,R=>R,C3=>C3,-- перенос
              F =>F,                -- управление
              D  =>D(12 downto 3),  -- входное данное
              CTR=>  A(12 downto 3)); -- выход счетчика
    A(2 downto 0)<=RG;
end PLM;
```

Эта архитектура была перекомпилирована в графическое представление структуры, которое показано на рис.4.5. Сравнение этой структуры со структурой на рис.4.4. указывает на их идентичность.

Испытательный стенд для ICTR

Рассмотрим испытательный стенд — объект ICTR_TB – для архитектуры ICTR(LUT), у которой эталонная модель - это архитектура ICTR(BEH). Испытания заключаются в подаче кода операции F с генератора

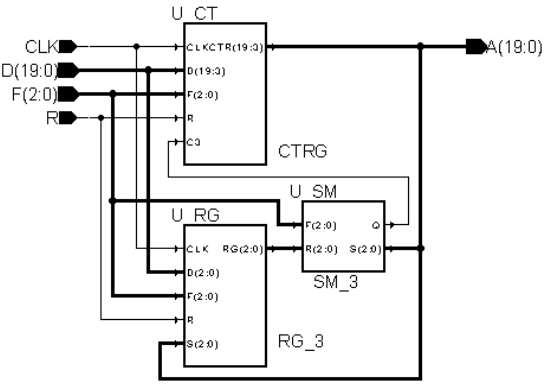


Рис.4.5. Структура ICTR, полученная из архитектуры ICTR(PLM)

случайных чисел на входы обеих моделей и сравнении выходных адресных последовательностей. Для такого сравнения используем следующий объект компаратора.

```
    entity COMPARATOR is
          generic(n:positive:=13;        -- разрядность векторов
       del:time:=20 ns);                --макс. задержка проверяемого у-ва
          port(CLK: in BIT;                --синхросерия
            D1 : in BIT_VECTOR(n-1 downto 0); --1-й вектор
                 D2 : in BIT_VECTOR(n-1 downto 0); --2-й вектор
                  Q : out BOOLEAN);    -- результат сравнения векторов
end COMPARATOR;
architecture BEH of COMPARATOR is
begin
          process  begin
                   wait until CLK='1';
                   wait for del;
                   Q<=true;
                   if  D1/=D2  then
                         Q<= false;
                   end if;
                   assert D1=D2 report "ошибка сравнения";
          end process;
    end BEH;
```

В нем с помощью настроечной константы n задается разрядность сравниваемых векторов. Поэтому он может быть использован в других проектах для сравнения векторов произвольной длины. Компаратор выдает булевский сигнал false, если векторы неодинаковые. При этом сравнение выполняется через del наносекунд после фронта синхросигнала, т.е. после того как прошли переходные процессы в тестируемых схемах. При несовпадении векторов на консоль симулятора также выдается сообщение об ошибке. Это сообщение выдается командой **assert**, которая является "ловушкой" ошибочных ситуаций при моделировании.

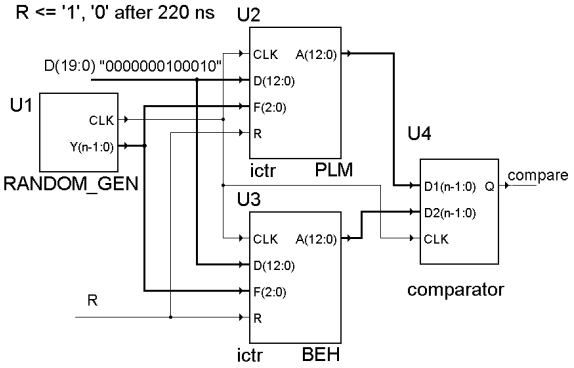


Рис. 4.6. Испытательный стенд для ICTR

В графическом виде испытательный стенд показан на рис. 4.6.

Вопросы по лабораторной работе.

Каково функциональное назначение ICTR?

Каким образом управляют режимом работы ICTR?

Для чего нужны константы **generic** в VHDL?

Покажите два способа вставки компонента в VHDL.

Объясните действие оператора **case**.

Объясните назначение оператора **assert**.

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

Какими способами можно выполнить описание устройства с триггерами?

Что выполняет функция конкатенации в VHDL?

Чем отличается описание аисхронных и синхронных триггеров?

Почему в проектах рекомендуют использовать только синхронные триггеры?

Предложите модель JK-триггера.

Предложите модель счетчика по модулю 1000.

Предложите модель счетчика по модулю 100 с начальной прегустановкой.

Разработайте модель сдвигового регистра с параллельным входом и последовательным выходом.

4.4. Оперативное запоминающее устройство

(Лабораторная работа 3)

Цель лабораторной работы

овладеть знаниями и навыками по проектированию устройств памяти, таких как ОЗУ (RAM). Лабораторная работа также служит для овладения навыками программирования и отладки описания RAM на языке VHDL.

Теоретические сведения

Блок RAM предназначен для быстрого доступа (Access) к данным, т.е. запоминания в блоке памяти (Memory) и выдачи n-разрядных слов по произвольному (Random) адресу. В работе предполагается, что данное имеет разрядность *n* от 4 до 16 бит и объем памяти равен *M* от 512 до 8192 слов. При проектировании микросхем, если объем RAM невелик (до 1024 бит), то память набирают из отдельных триггеров. В ином случае Пользуются библиоте-ками готовых модулей памяти. При разработке проекта для ПЛИС готовые модули памяти имеют объем 16, 32, 2048, 4096 бит. В последних сериях ПЛИС объем модуля RAM возрос до 16К или 18К бит. Разрядность модуля RAM может задаваться из ряда: 1,2,4,8 и 16 бит.

Запись данных и адреса в модуль RAM всегда выполняется по фронту синхросерии или сигнала записи, т.е. вход модуля можно рассматривать как вход синхронного регистра. Чтение данного чаще всего выполняется в следующем такте после такта приема адреса. Во многих случаях на выходе модуля RAM стоит синхронный регистр, запоминающий прочитанное слово. Запись и чтение из модуля может выполняться по конвейерному принципу: в одном такте записывается адрес нового данного и выдается прочитанное данное по предыдущему адресу. Для формирования RAM большого объема собирают систему из нескольких готовых модулей, дешифратора адреса для селекции модуля и выходного мультиплексора.

Различные варианты RAM в лабораторной работе имеют три, две или одну шину. В первом случае шины входного, выходного данного и адреса – раздельны, во втором случае шина входного и выходного данного совмещены и в третьем случае и адрес, и данные передаются по одной шине попеременно в разных тактах.

Примеры описания RAM

Рассмотрим пример проектирования RAM объемом 1024 шестнадцатиразрядных слов с одной шиной адреса и данных. Объявление объекта для такой RAM выглядит следующим образом.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_BIT.all;
library UNISIM;
use UNISIM.all;
```

```
entity RAM is   port(CLK : in BIT;  -- синхровход
          R : in BIT;  -- сброс
          WR: in BIT;  -- сигнал записи
          AE: in BIT;  -- сигнал фиксации адреса
          OE: in BIT;  -- сигнал выдачи прочитанного слова
          AD: inout STD_LOGIC_VECTOR(15 downto 0) );-- адрес/данное
end RAM;
```

Здесь сперва объявлено, что используются стандартная библиотека IEEE и библиотека моделей компонентов фирмы Xilinx. Из первой библиотеки берется тип STD_LOGIC и STD_LOGIC_VECTOR, необходимые для организации тристабильной шины и связи с библиотечными компонентами, функции преобразования из типа STD_LOGIC в тип BIT и обратно, а также функцию преобразования в целое. Из второй библиотеки берутся компоненты модулей RAM и тристабильного буфера.

Поведенческая модель RAM

Поведенческая модель RAM представляет собой процесс, который не особо отличается от процесса, описывающего регистр, но в котором вместо вектора битов применяется массив векторов.

```
architecture BEH of RAM is
      type MEM1KX16 is array(0 to 1023) of BIT_VECTOR(15 downto 0);
      constant RAM_init: MEM1KX16:=  --начальное состояние памяти
               (X"0000",X"0000",X"0000",X"0000",X"0000",X"0000", others=>
      X"0000");
      signal addr,do: BIT_VECTOR(15 downto 0);
      signal addri:natural;
begin

RG_ADDR:process(CLK,R) begin  --регистр адреса
    if R='1' then
        addr<=X"0000";
    elsif CLK='1' and CLK'event and AE='1' then
        addr<= To_BITvector(AD);
    end if;
 end process;
  ------ собственно блок памяти ------------------
RAM1K:process(CLK,addr,addri)
     variable RAM: MEM1KX16:= RAM_init;
     variable addri:natural;
   begin
     addri<= To_INTEGER(unSIGNED(addr(9 downto 0)));
     if CLK='1' and CLK'event then
          if WR = '1' then
             RAM(addri):= To_BITvector(AD);  -- запись
          end if;
          if R='1' then
            do<= X"0000";
          else
```

```
            do<= RAM(addri);
          end if;
      end if;
    end process;
  TRI:AD<= To_StdLogicVector(do) when OE='1' -- тристабильный выходной буфер
        else "ZZZZZZZZZZZZZZZZ";
  end BEH;
```

Для задания памяти требуемого объема был введен тип MEM1KX16, представляющий собой массив 1К шестнадцатиразрядных слов. Константа RAM_init этого типа представляет начальное состояние RAM. Если блок RAM предназначен для хранения программ, то коды этой константы могут быть кодами команд программы, исполняемой после включения процессора.

В процессе RG_ADDR описан регистр адреса, в который записывается адрес чтения или записи с общей входной-выходной шины по сигналу AE. Переменная RAM типа MEM1KX16 является основой блока RAM. Доступ к ячейкам памяти – элементам массива RAM – выполняется по значению целой переменной addri. Эта переменная получается из вектора адреса addr после преобразования его типа. Так как элементы – битовые векторы, а входные и выходные данные типа STD_LOGIC_VECTOR, то для их преобразования используются функции To_BITvector и To_StdLogicVector из библиотеки IEEE. Как и в блоках памяти ПЛИС, по сигналу R блок RAM выдает нулевое значение. В операторе с меткой TRI описан выходной буфер с тремя состояниями, который по сигналу OE выдает считанное данной, а иначе – вектор из элементов, означающих третье состояние.

Данная программа относится к моделям, описанным синтезируемым стилем. Но компилятор-синтезатор может выполнить память на отдельных триггерах, что влечет за собой большие аппаратные затраты.

Структурная модель RAM

Рассмотрим проектирование RAM на базе PLMT и блоков памяти RAMB4_S16. Блок RAM должен содержать четыре блока памяти RAMB, мультиплексор MUX считанных данных, регистр адреса RGA выходной тристабильный буфер (рис. 4.7.).

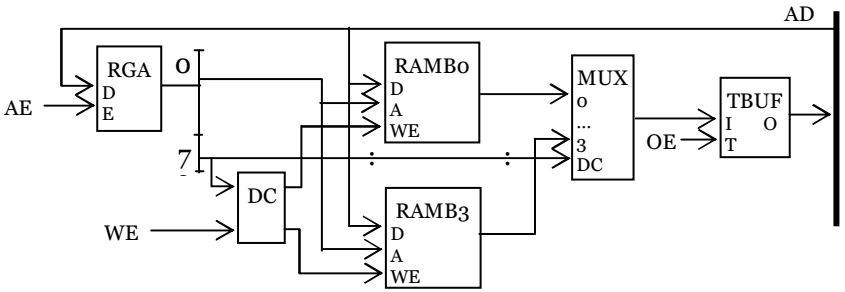


Рис.4.7. Структура блока RAM

Дешифратор DC двух старших разрядов адреса выдает отдельный сигнал записи на каждый из блоков RAMB. Дешифратор DC состоит из четырех PLM, каждая из которых декодирует 2 разряда адреса. Ниже описывается первая из них при C=WE, B=A_9, A=A_8.

```
architecture PLM_DC0 of PLM_3 is
begin
      Y<=(C and not B and not A) after td;
end PLM_DC0;
```

Остальные PLM дешифратора - PLM_DC1, PLM_DC2, PLM_DC3- описываются аналогично. Мультиплексор MUX состоит из шестнадцати PLM со следующим описанием при F=D_3, E=D_2, D = D_1, C = D_0, B=A_1, A=A_0.

```
architecture PLM_MUX of PLM_6 is
begin
      Y<=(C and not B and not A) -- 0-й вход
           or (D and not B and A)  -- 1-й вход
          or (E and B and not A)    -- 2-й вход
           or (F and B and A)          -- 3-й вход
       after td; --задержка элемента
end PLM_MUX;
```

Структурная модель RAM описывается как следующая архитектура.

```
architecture STR of RAM is
      type DARR_STD is array(0 to 3) of STD_LOGIC_VECTOR(15 downto 0);
      signal do_std:DARR_STD;     --выходы RAM
      constant one:STD_LOGIC:='1';
      constant gnd:BIT:='0';  -- нулевой бит
      signal DO0,DO1,DO2,DO3:BIT_VECTOR(15 downto 0);--выходы RAM
      signal CLK_std,rst,oe_std: STD_LOGIC;
      signal we:BIT_VECTOR(3 downto 0);
      signal we_std:STD_LOGIC_VECTOR(3 downto 0);
      signal addr: BIT_VECTOR(9 downto 0);
      signal addr_std:STD_LOGIC_VECTOR(9 downto 0);
      signal dom,adi: BIT_VECTOR(15 downto 0);
      signal dom_std:STD_LOGIC_VECTOR(15 downto 0);

      component RAMB4_S16 is                              -- блок памяти
            port (DI: in STD_LOGIC_VECTOR (15 downto 0);--входное данное
                EN  : in STD_ULOGIC;  -- разрешение обращения
                WE  : in STD_ULOGIC;  -- сигнал записи
                RST : in STD_ULOGIC;  -- асинхронный сброс
                CLK : in STD_ULOGIC;  -- сигнал синхросерии
                ADDR: in STD_LOGIC_VECTOR (7 downto 0);--адрес
                DO  : out STD_LOGIC_VECTOR (15 downto 0) );--выходное данное
      end component;

      component BUFT is                                  --тристабильный буфер
          port( O: out   STD_ULOGIC; -- выход
```

```
            I: in    STD_ULOGIC; -- вход
            T: in    STD_ULOGIC);-- управляющий вход,  если=1, то Z
    end component;
    component FDRE  is port (Q:out  BIT; --триггер
            D,C,CE,R  :in   BIT);
    end component;

begin
      adi<=To_BITVector(AD);
RG_A:for i in 0 to 9 generate                    --регистр адреса ----------
   U_RGA: FDRE port map(C=>CLK,R=>R,CE=>AE,D=>adi(i),Q=>addr(i));
  end generate;
-- Дешифратор адреса --------------------------------
U_DC0: entity PLM_3(PLM_DC0)
    port map(C=>WR,B=>addr(9), A=>addr(8),Y=>we(0));
U_DC1: entity PLM_3(PLM_DC1)
    port map(C=>WR,B=>addr(9), A=>addr(8),Y=>we(1));
U_DC2: entity PLM_3(PLM_DC2)
    port map(C=>WR,B=>addr(9), A=>addr(8),Y=>we(2));
U_DC3: entity PLM_3(PLM_DC3)
    port map(C=>WR,B=>addr(9), A=>addr(8),Y=>we(3));
   CLK_std<= To_StdULogic(CLK); RST<= To_StdULogic(R);
   addr_std<= To_StdLogicVector(addr);
RAMS: for i in 0 to 3 generate  -- блоки памяти ----------
      wr_std(i)<= To_StdULogic(WR(i));
  U_RAM: RAMB4_S16 port map (CLK=>CLK_std,RST=>rst, EN=>one,
       WE  =>WR(i),
       DI=>AD,                          --входное данное
       ADDR=> addr_std(7 downto 0),--адрес
       DO  => do_std(i));                --выходное данное
  end generate;
 do0<= To_BITVector(do_std(0));    do1<= To_BITVector(do_std(1));
 do2<= To_BITVector(do_std(2));    do3<= To_BITVector(do_std(3));
 oe_std<= not To_StdULogic(OE);
RAM_ENV:for i in 0 to 15 generate
 U_MUX: entity PLM_6(PLM_MUX)        -- Мультиплексор ----------
    port map (F=>do3(i),E=>do2(i),D=>do1(i),C=>do0(i),
          B=>addr(9),A=>addr(8),Y=>dom(i));
    dom_std(i)<= To_StdULogic(dom(i));
 U_TRI: BUFT                              -- Тристабильный буфер ----------
  port map(T=> oe_std,I => dom_std(i), O=>AD(i));
 end generate;
end STR;
```

Поскольку в лабораторных работах битовый тип – основной, а вставляемые компоненты – с портами типа Std_Logic, то в описании RAM применяются функции преобразования типов To_StdULogic, To_BITVector из битового типа в Std_Logic и обратно.

Испытательный стенд для RAM

Рассмотрим испытательный стенд - объект RAM_TB - для архитектуры RAM(STR), у которой эталонная модель - архитектура RAM(BEH). При испытаниях на обе модели подаются случайные числа и сигналы управления и сравниваются состояния общих шин моделей. При этом числа AD поступают с генератора случайных чисел на входы обеих моделей через тристабильные буферы. Для сравнения результатов моделирования используется такой же объект компаратора, как и в лабораторной работе 2 (см. рис.4.8).

```
r<='1', '0' after 220 ns;
wr<= '1' when dc(1 downto 0)="00" else '0';
ae<= '1' when dc(1 downto 0)="01" else '0';
oe<= '1' when dc(1 downto 0)="10" else '0';
AD1<= To_StdLogicvector(di) when oe='0' else (others=>'Z');
AD2<= To_StdLogicvector(di) when oe='0' else (others=>'Z');
```

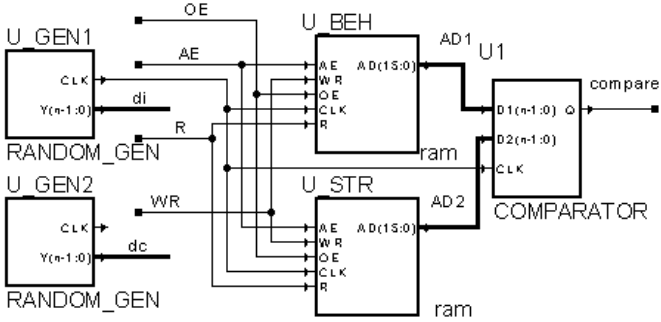


Рис. 4.8. Испытательный стенд для RAM

Вопросы к лабораторной работе.

Каково функциональное назначение RAM?
Каким образом управляют режимом работы RAM?
Какими способами задают RAM в проектах на VHDL?
Как проектировать RAM небольшого объема?
Как проектировать RAM объемом больше 1024 бит?
Для чего нужен тип std_ulogic?
Объясните работу тристабильной шины.
Покажите два способа задания тристабильного буфера в VHDL.
Как соединять порты и сигналы, которые имеют различные типы?
Почему рекомендуется все триггеры подключать к одному источнику синхросерии?

4.5. Регистровая память
(Лабораторная работа 4)

Цель лабораторной работы:

овладеть знаниями и практическими навыками по проектированию устройств памяти, таких как регистровое ОЗУ (FM). Лабораторная работа также служит для получения навыков программирования и отладки описания RAM на языке VHDL.

Теоретические сведения

Блок FM (Fast Memory – быстрая память) предназначен для быстрого доступа к N_R -разрядным словам по нескольким произвольным адресам. В лабораторной работе предполагается, что N_R изменяется от 4 до 16 бит и объем памяти M равен от 8 до 32 регистров. Каждый из каналов доступа к FM имеет свою адресную шину. Количество каналов доступа равно 2 или 3 и они обозначаются буквами B,D и Q. Один из каналов предназначен для записи, а остальные – для чтения. Каналы могут быть двунаправленными, т.е. использоваться как для записи, так и для чтения. В лабораторной работе двунаправленный канал рекомендуется исполнить на основе шины с тремя состояниями, как в лабораторной работе 3.

При проектировании микросхем, если объем RAM невелик, как в случае FM, то память набирают из отдельных триггеров. Запись данных в модуль FM всегда выполняется по фронту синхросерии или сигнала записи, т.е. вход модуля можно рассматривать как вход синхронного регистра. Прочитанное данное выдается сразу же после подачи адреса в FM.

Примеры описания FM

Рассмотрим пример проектирования трехканальной FM объемом 8 шестнадцатиразрядных слов. Такая FM может иметь такое объявление объекта:

```
use work.CNetwork.all;
entity FM is          port(CLK:in BIT; -- синхровход
    WR:in BIT;  -- сигнал записи
    AB:in BIT_VECTOR(2 downto 0);-- адрес канала B
    AD:in BIT_VECTOR(2 downto 0);-- адрес канала D
    AQ:in BIT_VECTOR(2 downto 0);-- адрес канала Q
    Q: in BIT_VECTOR (15 downto 0);-- данное канала Q
    B: out BIT_VECTOR(15 downto 0);-- данное канала B
    D: out BIT_VECTOR(15 downto 0));-- данное канала D
    end FM;
```

Поведенческая модель FM

Поведенческая модель FM во многом похожа на поведенческую модель RAM, описанную в лабораторной работе 3. Отличия заключаются в том, что две параллельных операции чтения и запись выполняются по трем различным адресам, а регистр адреса и тристабильный буфер - отсутствуют.

```
architecture BEH of FM is
type MEM8X16 is array(0 to 7) of BIT_VECTOR(15 downto 0);
signal addr,do: BIT_VECTOR(15 downto 0);
begin
FM8:process(CLK,AD,AB) ---- блок регистровой памяти ----------
    variable RAM: MEM8x16;
    variable addrq,addrd,addrb:natural;
  begin
    addrq:= BIT_TO_INT(AQ);  addrd:= BIT_TO_INT(AD);
    addrb:= BIT_TO_INT(AB);
    if CLK='1' and CLK'event then
        if WR = '1' then
          RAM(addrq):= Q;  -- запись
        end if;
    end if;
    B<= RAM(addrb);        -- чтение канала B
    D<= RAM(addrd);        -- чтение канала D
  end process;
end BEH;
```

Данная программа относится к моделям, описанным синтезируемым стилем. И компилятор - синтезатор выполняет эту память на отдельных триггерах.

Структурная модель FM

Рассмотрим проектирование FM на базе PLMT и триггеров. Блок FM должен содержать 8 регистров размером 16 бит, 2 мультиплексора считанных данных по каналам B, D и дешифратор записи по каналу Q (рис. 4.9).

Дешифратор DC состоит из восьми PLM, каждая из которых декодирует три разряда адреса. Первая из них описывается следующим образом при D=WE, C=A_2, B= A_1, A=A_0.

```
architecture PLM_DC0 of PLM_4 is
begin
Y<=(D and not C and not B and not A) after td;
end PLM_DC0;
```

Остальные PLM дешифратора - PLM_DC1, … , PLM_DC7- описываются аналогично. Каждый из мультиплексоров состоит из шестнадцати однобитных восьмивходовых мультиплексоров. Восьмивходовый мультиплексор представляется как один двухвходовой и два четырехвходовых мультиплексора. Четырехвходовый мультиплексор описан в

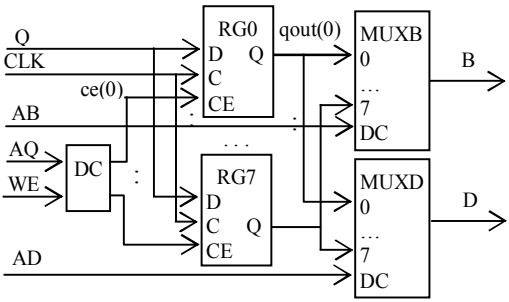


Рис.4.9. Структура блока FM

лабораторной работе 3 как PLM с архитектурой PLM_6(PLM_MUX). Двухвходовый мультиплексор имеет следующую архитектуру при C = D_1, B= D_0, A=A_0.

```
architecture PLM_MUX of PLM_3 is
begin
      Y<=(B and not A)  -- 0-й вход
           or (C and A)    -- 1-й вход
      after td;  --задержка элемента
end PLM_MUX;
```

Структурная модель восьмивходового мультиплексора описывается в следующем объекте.

```
entity MUX8 is port(D0,D1,D2,D3,D4,D5,D6,D7: in BIT; -- входы данных
                A: in BIT_VECTOR(2 downto 0); -- адрес
                Q: out BIT);                  -- выход данного
end MUX8;
architecture STR of MUX8 is
      signal mux0,mux1:BIT;
 begin
  U_MUX0: entity PLM_6(PLM_MUX)
       port map(F=>D3,E=>D2,D=>D1,C=>D0,B=>A(1),A=>A(0),Y=>mux0);
  U_MUX1: entity PLM_6(PLM_MUX)
      port map(F=>D7,E=>D6,D=>D5,C=>D4,B=>A(1),A=>A(0),Y=>mux1);
  U_MUX3: entity PLM_3(PLM_MUX)
      port map(C=>mux1,B=>mux0,A=>A(2),Y=>Q);
 end STR;
```

Тело архитектуры регистровой памяти выглядит так.

```
architecture STR of FM is
      type FMARR is array(7 downto 0, 15 downto 0) of BIT;
      signal y:FMARR;
      signal ce:BIT_VECTOR(7 downto 0);
      constant gnd:BIT:='0';
      component MUX8 is port(D0,D1,D2,D3,D4,D5,D6,D7: in BIT; -- входы
                  A: in BIT_VECTOR(2 downto 0); -- адрес
                  Q: out BIT);                  -- выход данного
      end component;
      component FDRE  is port (Q:out  BIT; --триггер
                               D, C,CE,R: in   BIT );
      end component;
begin            -- дешифратор адреса  (компоненты U_DC2,..., U_DC6 опущены)
     U_DC0: entity PLM_4(PLM_DC0)
       port map(D=>WR,C=>AQ(2),B=>AQ(1),A=>AQ(0),Y=>ce(0));
     . . .
     U_DC7: entity PLM_4(PLM_DC7)
       port map(D=>WR,C=>AQ(2),B=>AQ(1),A=>AQ(0),Y=>ce(7));
  -- массив регистров  ----------------------
  U_FM: for i in 0 to 7 generate
```

```
        U_RG: for j in 0 to 15 generate
                U_TT: FDRE port map (D=>Q(j),  -- входное данное
                        C =>CLK,   -- синхросигнал
                        CE=> ce(i),-- разрешение записи
                        R => gnd,  -- сброс не используется
                        Q=>y(i,j)); -- выходы триггеров
        end generate;
  end generate;
-- выходные мультиплексоры  ----------
U_MUX: for i in 0 to 15 generate
    MUXD:MUX8 port map(D0=>y(0,i),D1=>y(1,i),D2=>y(2,i),D3=>y(3,i),
            D4=>y(4,i),D5=>y(5,i),D6=>y(6,i),D7=>y(7,i),
            A=>AD,       -- адрес
            Q=>D(i));    -- выход канала D
    MUXB:MUX8 port map(D0=>y(0,i),D1=>y(1,i),D2=>y(2,i),D3=>y(3,i),
            D4=>y(4,i),D5=>y(5,i),D6=>y(6,i),D7=>y(7,i),
            A=>AB,       -- адрес
            Q=>B(i));    -- выход канала B
  end generate;
end STR;
```

Для связи выходов триггеров FM с входами мультиплексоров используется сигнал y типа двумерный массив размерами 8x16.

Испытательный стенд для FM

В испытательном стенде - объекте FM_TB – проверяется архитектура FM(STR), у которой эталонной модель - архитектура RAM(BEH). Случайные адреса, данные и периодический сигнал записи подаются на входы обеих моделей, а состояния выходных шин данных сравниваются между собой. Испытательный стенд аналогичен стенду в лабораторной работе 3. Но количество генераторов случайных чисел увеличено до 4.

Вопросы к лабораторной работе.

Каково функциональное назначение FM?

Какие элементы используют при проектировании FM?

Покажите 5 способов задания дешифратора на 4 выхода.

Какими способами задают FM в проектах на VHDL?

От чего и насколько зависят аппаратные затраты FM?

Почему в структурном проекте FM вставка дешифратора выполнена как вставка объекта, а вставка триггера – как вставка компонента?

Почему все триггеры FM надо подключать к одному источнику синхросерии?

4.6. Блок умножения

(Лабораторная работа 5)

Цель лабораторной работы

овладеть знаниями и практическими навыками по проектированию вычислительных блоков последовательного действия, таких как блок умножения (MPU), включая его управляющий автомат (finite state mashine - FSM). Также даются навыки программирования и отладки описания блоков последовательного действия и FSM на языке VHDL.

Теоретические сведения

Блоки умножения и деления последовательного действия применяются в простейших процессорах для выполнения соответствующих команд. Они имеют в несколько раз меньшие аппаратурные затраты, чем блоки параллельного умножения и деления. При умножении двух n-разрядных операндов получают $2n$ – разрядное произведение. Исходными данными для деления обычно являются $2n$ – разрядное делимое и n - разрядный делитель. Результаты - n - разрядное частное и n – разрядный остаток. Различают операции умножения и деления с учетом и без учета знаков операндов.

Последовательные операции умножения и деления связаны со сдвигом одного или двух операндов и/или частного результата. В зависимости от выполнения тех или иных регистров блока регистрами сдвига, различают 4 основных схемы умножения и 2 схемы деления. Кроме того, блоки деления могут выполнять схему деления без восстановления и с восстановлением остатка.

Некоторые блоки для вычисления элементарных функций, например, выполняющие алгоритмы "цифра за цифрой", работают аналогично последовательному блоку умножения.

Пример описания MPU

Рассмотрим пример проектирования 16- разрядного MPU, выполняющего алгоритм умножения в дополнительном коде, начиная с младших разрядов множителя с неподвижным множимым и полным результатом. Тогда операция умножения имеет 2 операнда, поступающих по двум шинам и 2 шестнадцатиразрядных результата (старшая и младшая части произведения), выдаваемые последовательно через одну шину результата. Интерфейс блока:

```
entity MPU is
        port(CLK : in BIT;
             RST : in BIT;
             START:in BIT;-- пуск умножения по фронту
             OUTHL:in BIT;-- выдача старшего(0) или младшего(1) слова P
             DA : in BIT_VECTOR(15 downto 0);  -- операнд A
             DB : in BIT_VECTOR(15 downto 0);  -- операнд B
             RDY : out BIT;    -- результат готов
             Z: out BIT;       -- признак нулевого результата
```

```
        N: out BIT;       -- 1 - отрицательный результат
        DP : out BIT_VECTOR(15 downto 0) );-- слово результата P
end MPU;
```

Блок состоит из операционной части и управляющего автомата. Операционная часть имеет структуру, представленную на рис.4.10. Она описана в следующем объекте:

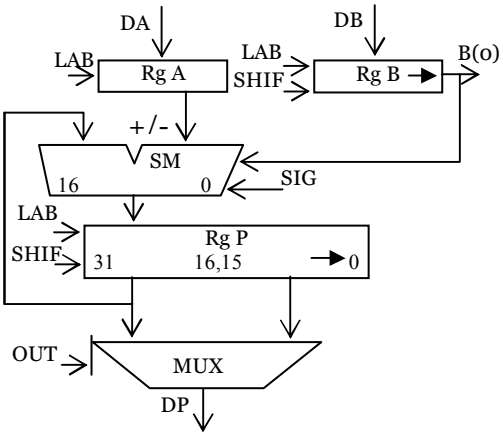


Рис.4.10. Структура операционной части блока

```
library IEEE;
use IEEE.Numeric_BIT.all;
entity MPU_OP is
   port(C : in BIT;
    RST : in BIT;
    LAB : in BIT;  -- загрузка A и B
                   --  и обнуление P
    SHIFT : in BIT;  --разрешение
                     -- сдвига A и P
    SIGN: in BIT; -- знак сложения
         -- частных произведений
    OUTHL : in BIT;  --выдача
              --старшего (0) или
              --младшего (1) слова
                DA : in BIT_VECTOR(15 downto 0);
                DB : in BIT_VECTOR(15 downto 0);
                Z: out BIT;          -- признак нулевого результата
                N: out BIT;          -- 1 - отрицательный результат
                DP : out BIT_VECTOR(15 downto 0)); -- шина произведения
end MPU_OP;
architecture BEH of MPU_OP is
      signal A,B: BIT_VECTOR(15 downto 0); -- операнды A и B
      signal S: SIGNED(16 downto 0);   -- сумма частных произведений
      signal P: SIGNED(31 downto 0);   -- произведение
begin
      RG_A:process(C,RST)    -- регистр операнда A
      begin
            if RST='1' then
                  A<=X"0000";
            elsif C='1' and C'event then
                  if LAB='1' then
                        A<=DA;   -- запись в регистр
                  end if;
            end if;
      end process;
      RG_B:process(C,RST) begin     -- регистр операнда B
            if RST='1' then
               B<=X"0000";
            elsif C='1' and C'event then
               if LAB='1' then
```

```
                    B<=DB;                          -- запись в регистр
                elsif SHIFT='1' then
                    B<='0'& B(15 downto 1);     --сдвиг регистра
                end if;
            end if;
    end process;
-- Сумматор - вычитатель частных произведений
    SM:S<= P(31)&P(31 downto 16)-SIGNED(A) when B(0)='1' and SIGN='1'
            else P(31)&P(31 downto 16)+SIGNED(A) when B(0)='1'
            else P(31)&P(31 downto 16);
    RG_P:process(C,RST,P)    -- регистр P
        variable zi:BIT;
    begin
            if RST='1' then
                P<=X"00000000";
            elsif C='1' and C'event then
                if LAB='1' then
                    P<=X"00000000";     -- запись в регистр  0
                elsif SHIFT='1' then
                    P<=S & P(15 downto 1);      --сдвиг регистра
                end if;
            end if;
            zi:='0';      -- цикл определения нулевого результата
            for i in P'range loop
                zi:=zi or P(i);
            end loop;
            Z<= not zi;             -- флаг нулевого произведения
            N<=P(31);               -- флаг знака
    end process;

-- выходной мультиплексор  ------------------------------------
    MUX_P:DP<= BIT_VECTOR(P(15 downto 0)) when OUTHL='1' else
                BIT_VECTOR(P(31 downto 16));
end BEH;
```

В процессах RG_A, RG_B, RG_P описаны регистры операндов и произведения. Оператором SM представлен сумматор-вычитатель, который в зависимости от очередного бита множителя B(0) прибавляет или нет к сумме частных произведений P множимое A. В последнем такте умножения, когда выполняется умножение на знаковый разряд, т.е. при SIGN='1' , из P вычитается множимое A.

Признак нулевого результата Z вычисляется в 32- разрядной схеме ИЛИ-НЕ, которая реализована в операторе **loop**. Выходной мультиплексор MUX_P выдает старшее или младшее слово произведения в зависимости от сигнала OUTHL.

Управляющий автомат для блока умножения описан в следующей архитектуре.

```
    entity MPU_FSM is port(C : in BIT;
                    RST : in BIT;
                    START : in BIT;   -- начать умножение
                    LAB : out BIT;    -- загрузка A и B и обнуление P
                    SHIFT : out BIT;  --разрешение сдвига A и P
                    SIGN: out BIT;    -- знак сложения
                    RDY : out BIT);   --конец умножения
end MPU_FSM;
architecture BEH of MPU_FSM is
        type STATES is (s1,s2,s3,s4,s5,s6,s7,s8,s9,s10, --состояния автомата
        s11,s12,s13,s14,s15,s16,s17,finish);
        signal st:STATES;
begin
        STATE:process(C,RST)          -- регистр состояния
        begin
            if RST='1' then
                st<=finish;
            elsif C='1' and C'event then
                case st is                    -- определение следующего состояния
                  when s1=> st<=s2;
                  when s2=> st<=s3;   --остальные состояния - аналогично
                            . . .
                  when s16=> st<=s17;
                  when s17=> st<=finish;
                  when finish=> if START ='1' then --последнее состояние
                                    st<=s1;
                                else
                                    st<=finish;
                                end if;
               end case;
                end if;
        end process;
-- логика выходных сигналов
        LAB<='1' when st=s1 else '0';
        SHIFT<='0' when st=s1 or st=finish else '1';
        SIGN<='1' when st=s17 else '0';
        RDY<='1' when st=finish else '0';
    end BEH;
```

Автомат имеет 18 состояний, заданных типом. В операторе STATE описан регистр состояния st и функция изменения состояния, заданная оператором **case**. Каждая строка оператора **case** определяет, какое новое состояние примет регистр состояния, если автомат находится в данном состоянии (несколько строк заменено многоточием). Начальное состояние автомата управления умножением после сброса - finish. Далее состояния автомата изменяются циклически, и он останавливается в состоянии finish с циклом ожидания прихода сигнала START.

Параллельные операторы присваивания задают логику выходных сигналов автомата, которая зависит от его текущего состояния. Таким образом,

данный автомат – это автомат Мура. Архитектура блока умножения описана ниже.

```
architecture BEH of MPU is
          component MPU_OP is port(C, RST : in BIT;
               LAB : in BIT;  -- загрузка A и B и обнуление P
               SHIFT : in BIT;  --разрешение сдвига A и P
               SIGN: in BIT;  -- знак сложения
               OUTHL : in BIT;  --выдача старшего(0) или младшего(1) слова
               DA , DB : in BIT_VECTOR(15 downto 0);
               Z: out BIT;              -- признак нулевого результата
               N: out BIT;              -- 1 - отрицательный результат
               DP : out BIT_VECTOR(15 downto 0)); -- шина произведения
          end component;
          component MPU_FSM is port(C, RST: in BIT;-- управляющий автомат
               START : in BIT; -- начать умножение
               LAB : out BIT;   -- загрузка A и B и обнуление P
               SHIFT : out BIT; --разрешение сдвига A и P
               SIGN: out BIT;   -- знак сложения
               RDY : out BIT);  --конец умножения
          end component ;
          signal lab,shift,sign:BIT;
begin
-- Операционная часть--------
U_OP:MPU_OP   port map(C,RST,
          LAB=>lab, SHIFT=>shift,
          SIGN=>sign,
          OUTHL=>OUTHL,
          DA=>DA, DB=>DB,
           Z=>Z, N=>N,
             DP=>DP);
-- Управляющий автомат --
   U_FSM:MPU_FSM  port map(C,RST,
          START=>START,
          LAB=>lab,
          SHIFT=>shift,
          SIGN=>sign,
          RDY=>RDY);
end BEH;
```

Структура блока умножения, полученная с помощью утилиты Code2Graphics, показана на рис. 4.11.

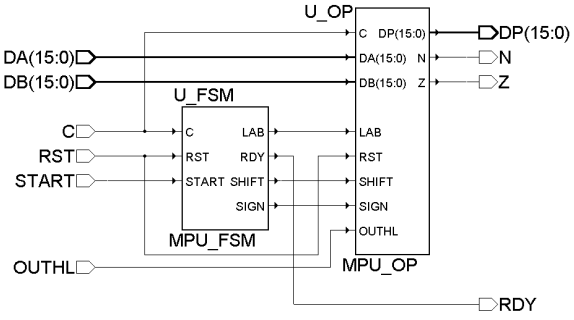


Рис.4.11. Структура блока умножения

Испытательный стенд для MPU

Испытательный стенд для MPU включает в себя контролируемый объект – блок умножения, источники тестовых сигналов и процесс контроля результатов умножения. Его описательная часть представлена ниже.

```
begin
        c<=not c after 5 ns;  --генератор синхросигнала с периодом 10 нс
        rst<='1', '0' after 33 ns;        -- генератор сигнала сброса
        DA<=X"1234", X"8910" after 400 ns;    -- подача операнда A
        DB<=X"f234", X"7654" after 400 ns;    --  подача операнда B
        OUTHL<='1' after 350 ns, '0' after 450 ns, '1' after 790 ns;
        START<= '1' after 40 ns, '0' after 50 ns, '1' after 500 ns, '0' after 600 ns;
UUT : mpu port map (C => C,RST => RST,--тестируемое устройство
                          START => START,   OUTHL => OUTHL,
                          DA => DA,DB => DB,
                          RDY => RDY,   Z => Z,
                          N => N,   DP => DP);
TST:process begin  -- проверка результатов умножения
                                wait for 300 ns;
                    Assert DP=X"FF04" report "Ошибка в старшем слове P";
                                wait for 100 ns;
                    Assert DP=X"DA90" report "Ошибка в младшем слове P";
                                wait for 300 ns;
                    Assert DP=X"C906" report "Ошибка в старшем слове P";
                                wait for 100 ns;
                    Assert DP=X"5940" report "Ошибка в младшем слове P";
                                wait;
    end process;
end TB_ARCHITECTURE;
```

Правильность выполнения алгоритма умножения проверяется в процессе TST. При этом через соответствующие промежутки времени состояние шины сравнивается с ожидаемым результатом и в случае несовпадения на консоль выдается сообщение об ошибке. Также правильность выходных сигналов можно проверить, исследуя их графики, а также состояние st его управляющего автомата, которые представлены на рис. 4.12.

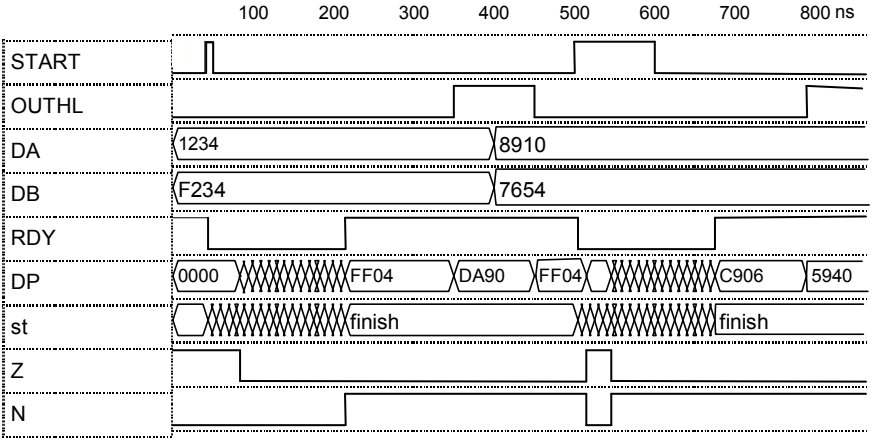


Рис.4.12. Графики входных и выходных сигналов блока умножения.

Вопросы по лабораторной работе.

Какие алгоритмы последовательного умножения и деления вы знаете?

Что такое алгоритм "цифра за цифрой" и как он связан с алгоритмом последовательного умножения?

Почему вычислительное устройство проектируют в виде совокупности операционного и управляющего автоматов?

Для чего при описании сумматора применяют подтип SIGNED?

Приведите 2 способа программирования регистра сдвига в VHDL.

Приведите пример программирования n-разрядной схемы И в VHDL.

Приведите 4 способа программирования мультиплексора в VHDL.

Нарисуйте граф-схему алгоритма управляющего автомата блока умножения.

Опишите на VHDL управляющий автомат блока умножения, выполненный на основе счетчика.

Опишите на VHDL управляющий автомат блока умножения, выполненный на основе сдвигового регистра.

Опишите на VHDL блок умножения, выполненный на основе параллельного умножителя.

Опишите на VHDL блок умножения 2n-разрядных чисел, выполненный на основе n-разрядных параллельных умножителей.

4.7 Арифметическое устройство

(Лабораторная работа 6)

Цель лабораторной работы:

овладеть знаниями и практическими навыками по проектированию вычислительных блоков, таких как арифметическое устройство (AU).

Теоретические сведения

Блок AU является центральной частью практически любого процессора и предназначен для выполнения арифметических и логических операций с операндами, заданными в текущей команде. Перечень операций и операндов определен в системе команд данного процессора.

Перечень операций AU определяет производительность процессора, с одной стороны и сложность AU и дешифратора команд, с другой стороны. Минимальный перечень операций включает в себя операции, обеспечивающие алгоритмическую полноту системы команд. Это, как минимум, три логические операции, например, И, НЕ, установка в 0, арифметические операции сложения, вычитания, а также операции сдвига на один разряд вправо, выполняемые с целыми числами. Остальные арифметические и логические функции вычисляются путем выполнения цепочек команд.

Для увеличения производительности процессора и сокращения длины программ в перечень команд вводятся операции вычисления функций, которые часто встречаются в реализуемых алгоритмах. Поэтому операции умножения и деления входят в систему команд большинства процессоров. Выбор перечня операций представляет собой поиск компромисса между удобством системы команд, объемом памяти, занимаемым программой, быстродействием процессора и сложностью AU.

Операнды в AU могут поступать из регистровой памяти FM, оперативной памяти RAM или из специальных регистров, например, из регистра–аккумулятора. Эти же блоки и регистры служат приемниками результата AU. Признак переноса C участвует в операциях как особенный операнд.

Источники операндов определяются видами адресации системы команд. При использовании нескольких видов адресации входы AU должны содержать мультиплексоры операндов, которые подключают те или иные источники операндов. Такие мультиплексоры вносят дополнительные аппаратные затраты и добавочную задержку в цикл выполнения команд.

В простейшем случае все операнды приходят из FM. Такая адресация операндов применяется в большинстве RISC – процессоров. Если FM и LSM реализованы в одном блоке AU, то большинство передач данных и арифметико-логических операций выполняется внутри блока, т.е. они - локальные. При этом достигается минимальная длительность командного цикла.

Признаки результатов выполнения операций обычно запоминаются в специальном регистре, который входит в состав регистра состояния про-

цессора. Кроме признака переноса C, знака результата N, нулевого результата Z, фиксируется также признак переполнения V, признаки исключительных состояний, например, деления на ноль, выход за допустимые границы адресного пространства и т.п. При операциях сдвига выдвигаемый разряд чаще всего запоминается в разряде признака переноса C.

Как правило, операция в AU выполняется за один такт, если операнды и результат хранятся в FM. Многие операции, например, операции ум-ножения, деления в простейших AU, операции вычисления элементарных функций, операции с плавающей запятой, выполняются за несколько тактов. При этом AU должно работать под управлением специального автомата.

Для повышения производительности процессора уменьшают длительность тактового интервала вставкой регистров промежуточных результатов. При этом в AU организуется конвейерная обработка операндов, а в процесссоре в целом - конвейерное исполнение команд. В конвейерном AU на соседних ступенях конвейера одновременно выполняются разные стадии обработки команд. Такое AU также должно обеспечивать заполнение и очистку ступеней конвейера при выполнении программного перехода, сохранение и восстановление его состояния при вызове подпрограммы и возврате из нее. Поэтому конвейерное AU намного сложнее, чем обычное AU.

Пример описания AU

Рассмотрим пример проектирования 16- разрядного AU, выполняющего набор операций, представленный в таблице 4.6 на базе трехкнального блока FM с восемью регистрами. В состав AU входит блок LSM, выполняющий операции сложения и вычитания с переносом, логические И и ИЛИ. Операции сложения и вычитания образуются путем подачи 0, 1 или флага переноса C на вход C0 блока LSM (см. табл.4.6).

Таблица 4.6. Кодирование операций ALU

Операция	Мнемо-ника	Код операции ACOP	Код F управления LSM	Бит переноса C0 или бит S15
Сложение	ADD	0000	00	0
Сложение с переносом	ADDC	0001	00	C
Сложение с инкрементом	ADDINC	0010	00	1
Вычитание с декрементом	SUBDEC	0100	01	0
Вычитание с переносом	SUBC	0101	01	C
Вычитание	SUB	0110	01	1
Логическое И	AND	1010	10	-
Логическое ИЛИ	OR	1110	11	-
Сдвиг вправо логический	SRL	1000	-	0
Сдвиг вправо с переносом	SRC	1001	-	C
Сдвиг вправо арифметический	SRA	1011	-	N
Умножение	MUL	1100	-	-

Для сдвигов вправо необходимо подключить сдвигатель на один разряд. При этом для выполнения логического, арифметического сдвигов или сдвига с переносом в левый разряд результата вставляется 0, флаг C или флаг знака N. Операция умножения выполняется последовательно-параллельно в блоке умножения, разработанном в лабораторной работе 5.

Поскольку блок FM – трехадресный, то по одной команде выбирается 2 источника операндов и 1 приемник – результат. Как видно из табл.4.7, всего использовано 12 комбинаций кодов операции, т.е. в систему операций AU при необходимости можно добавить ещё 4 операции.

Операция умножения имеет 2 операнда и 2 шестнадцатиразрядных результата (старшая и младшая части произведения). Так как задан один адрес результата, то целесообразно 2 слова результата располагать в двух соседних регистрах, адреса которых отличаются на 1. Для этого на адресном входе AQ блока FM следует вставить схему инкремента.

Все операции должны выдавать признаки результата C – бит переноса, Z – признак нулевого результата и бит знака N, которые должны запоминаться в регистре состояния. При выполнении вызова подпрограмм в области стека должно сохраняться состояние регистра состояния и адрес возврата из подпро-граммы, а по команде возврата из под-программы этот ад-рес возвращается в ICTR. В RISC – процесорах при вы-зове подпрограмм адрес возврата сохраняется в одном из регистров блока FM, например, в последнем. Так как в учебном процессоре адресное пространство ограничено 13 разрядами, то целесообразно младшие разряды регистра 7 отвести под адрес возврата, т.е. биты C, Z и N будут сохраняться в 15, 14 и 13 разрядах этого регистра, соответственно.

Таким образом, для реализации всех функций AU необходимо внести дополнения в блок FM, что приводит к изменению как его структуры, так и интерфейса. Структура блока FM показана на рис.4.13.

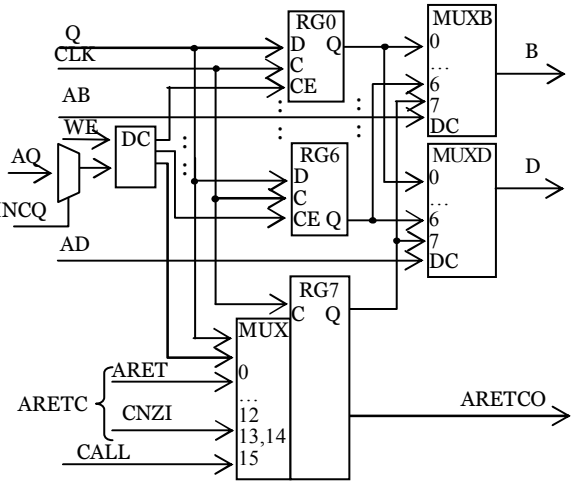


Рис.4.13. Структура модернизированного блока FM

Структура блока AU показана на рис. 4.14. В него кроме блока FM входят ранее разработанные блоки MPU и LSM. Мультиплексор MUXQ предназначен для подачи на выход DO блока и на вход записи Q входного данного, или результата LSM, или результата MPU, или считанного данного D, или его же, но сдвинутого вправо по командам SRL, SRC или SRA. Мультиплексор MUXC выбирает флаг переноса C или 0, или 1 для подачи на вход переноса LSM для выполнения команд сложения и вычитания.

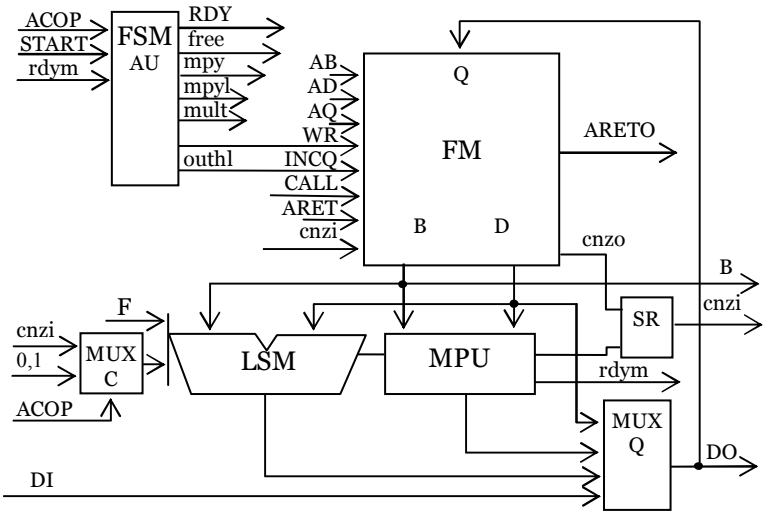


Рис.4.14. Структура блока AU

Регистр состояния SR записывает в конце выполнения команд признак переноса C, признак отрицательного результата N и признак нулевого результата Z, причем, если это команда умножения, то состояние записывается с блока умножения, если выполняется команда возврата из подпрограммы RET – то состояние, запомненное в 7-м регистре FM, а иначе – с блока LSM.

На выход BO подается данное, прочитанное из FM по адресу AB. Это данное может использоваться как адрес при индексной адресации внешней памяти.

Управляющий автомат FSM_AU имеет 3 состояния: free -AU свободен, mpy - идет умножение, mpyl - конец умножения, в зависимости от которых и входных сигналов ACOP, START и окончания умножения rdym формирует необходимые управляющие сигналы. Его диаграмма состояний показана на рис.4.15.

Рис.4.15. Диаграмма состояний FSM_AU

Интерфейс объекта AU имеет следующее описание.

```
entity AU is        port( CLK : in BIT;
                    RST : in BIT;
                    START : in BIT;     --начать операцию AU
                    RD: in BIT;         -- чтение из FM на шину DO
                    WRD : in BIT;       -- запись с шины DI
                    RET : in BIT;       -- возврат из подпрограммы
                    CALL: in BIT;       -- вызов подпрограммы
                    DI : in BIT_VECTOR(15 downto 0);   --вх. шина данных
                    AB : in BIT_VECTOR(2 downto 0);     -- адрес регистра B
                    AD : in BIT_VECTOR(2 downto 0);     -- адрес регистра D
                    AQ : in BIT_VECTOR(2 downto 0);     -- адрес регистра Q
                    ARET : in BIT_VECTOR(12 downto 0);  -- адрес возврата
                    ACOP : in BIT_VECTOR(3 downto 0);   -- код операции AU
                    RDY : out BIT;                      --готовность результата
                    ARETO : out BIT_VECTOR(12 downto 0);-- адрес возврата
                    DO : out BIT_VECTOR(15 downto 0); --вых. шина данных
          BO : out BIT_VECTOR(15 downto 0); --вых. шина данного B
                    CNZ: out BIT_VECTOR(2 downto 0)); --вых. рег. состояний
    end AU;
```

Описание архитектуры AU, в котором не приведены описания ее компонентов, выглядит следующим образом.

```
. . .
          type STAT_AU is (free,mpy,mpyl);-- состояния автомата
          signal st:STAT_AU;
          signal b,q,d,y,dp,aretc,aretco:BIT_VECTOR(15 downto 0);
          signal c0,c15,csh,zlsm,wr,mult,mstart,outhl:BIT;
          signal rdym,zmpy,nmpy:BIT;
          signal cnzr,cnzo,cnzi:BIT_VECTOR(2 downto 0);
begin
          U_FM: FM2 port map(CLK,    -- блок регистровой памяти
                    WR=>wr,  INCQ=>outhl, CALL=>CALL,
                    AB=>AB,  AD=>AD, AQ=>AQ,
                    ARETC=>aretc,
                    Q=>q,  B=>b, D=>d,
                    ARETCO=>aretco);
          aretc<=cnzr&ARET;
          cnzo<=aretco(15 downto 13);
          ARETO<=aretco(12 downto 0);
          MUX_C:c0<='1' when ACOP(1 downto 0)="10"else  --мультиплексор C0
                    cnzi(2) when ACOP(1 downto 0)="01"else '0';
          U_LSM:LSM port map(F=>ACOP(3 downto 2),  -- LSM
                    A=>d,     B=>b,
                    C0=>c0,  Y =>y,
                    C15=>c15,   Z =>zlsm);
          U_MPU:MPU port map(CLK,RST, -- блок умножения
                    START=>mstart,    OUTHL=>outhl,
                    DA=>d,            DB=>b,
                    RDY=>rdym,        Z=>zmpy,
                    N=>nmpy,          DP=>dp);
          MUX_CI:csh<=cnzi(2) when ACOP(1 downto 0)="01" else --задвиг. разр.
```

```
                cnzi(1) when ACOP(1 downto 0)="11" else  '0';
-----   мультиплексор результата ----------------------------------------
        MUX_Q:q<=dp when st/=free else                  --результат умножения
        csh&d(15 downto 1) when ACOP="1000"   -- сдвиг вправо
                or ACOP="1001" or ACOP="1011" else
             DI when WRD='1'   else                     --входное данное
             d when RD='1' else                         --данное из FM по адресу AD
             y;                                         --результат LSM
        SR:process(CLK,RST)             --регистр состояния с мультиплексором
        begin
                if RST='1' then
                        cnzi<="000";
                elsif CLK='1' and CLK'event then
                        if RET='1' then
                                cnzi<=cnzo;
                        elsif  st=mpyl then
                                cnzi<='0'&nmpy&zmpy;
                        elsif mult='0' then
                                cnzi<=c15&y(15)&zlsm;
                        end if;
                end if;
        end process;
        mult<='1' when ACOP="1100" else '0';--дешифрация умножения
        FSM_AU:process(CLK,RST)             --автомат управления
        begin
           if RST='1' then
              st<=free;                  -- регистр состояния автомата
           elsif CLK='1' and CLK'event then
              case st is
                 when free => if START='1'and  mult='1'then --свободен
                                st<=mpy;
                              end if;
                 when mpy=> if rdym='1' then -- идет умножение
                                st<=mpyl ;
                              end if;
                 when mpyl=>  st<=free; --    конец умножения
              end case;
           end if;
        end process;
--функции выходов автомата -------------------------
        outhl<='1' when st=mpyl else '0';
        wr<='1' when WRD='1' or st=mpyl or (st=mpy and rdym='1')
                 or (START='1' and mult='0')  else '0';
        mstart<='1' when mult='1' and (st/=mpy and st/=mpyl     )else'0'  ;
        RDY<='1' when st=mpyl or (WRD='0' and st/=mpy and mult='0')else'0';

        DO<=q; --выходное данное
  BO<=B;
        CNZ<=cnzi;          --выход регистра состояния
end BEH;
```

Испытательный стенд для AU

AU представляет собой сложный блок с внутренней памятью. Для его проверки предлагается испытательный стенд на основе простого устройства микропрограммного управления. Его описание архитектуры (кроме описания компонента AU и входов-выходов ALU) представлено ниже.

```
. . .
        type MICROINST is record                    -- формат микрокоманды
                ACOP:BIT_VECTOR(3 downto 0);        -- код операции AU
                AQ,AD,AB:BIT_VECTOR(2 downto 0);    --адреса FM
                DI:BIT_VECTOR(15 downto 0);         -- входное данное
                START,WRD,RD:BIT;                   -- биты управления
        end record;
        constant n: positive:=6;                --число микрокоманд
        type MICROPROGR is array(0 to n-1) of MICROINST;
        constant   mp:MICROPROGR:=(-- ПЗУ тестирующей микропрограммы
        ("0000","001","000","000",X"4000",'0','1','0'),      --L 1,#4000h
        ("0000","010","000","000",X"cfff",   '0','1','0'),    --L 2, #-3001h
        ("0010","011","001","010",X"0000",'1','0','0'),      --ADDINC 3,1,2
        ("0101","011","001","010",X"0000",'1','0','0'),      --SUBC 3,1,2
        ("1100","100","011","010",X"0000",'1','0','0'),      --MUL 4,3,2
        ("0000","000","100","000",X"0000",'0','0','1'));     --S 4
        signal maddr:natural;
begin
        CLK<=not CLK after 5 ns; -- генератор синхросигнала
        RST<='1' ,'0' after 25 ns; -- генератор сигнала сброса
        CTM:process(CLK,RST) begin          -- счетчик микрокоманд
                        if RST='1' then
                    maddr<=0;
                elsif CLK='1' and CLK'event then
                    if (RDY='1' and START='1') or WRD='1'or RD='1'  then
                            maddr<=(maddr+1) mod n; -- +1 к счетчику
                    end if;
                end if;
        end process;
ROM_MP:(ACOP,AQ,AD,AB,DI,START,WRD,RD)<=mp(maddr);
UUT : AU port map (CLK,RST,         --тестируемое AU
                START => START,
                RD => RD,
                WRD => WRD,
                RET => RET,
                CALL => CALL,
                DI => DI,
                AB => AB,
                AD => AD,
                AQ => AQ,
                ARET => ARET,
                ACOP => ACOP,
                RDY => RDY,
                ARETO => ARETO,
```

```
                DO => DO,
                BO=>BO,
                CNZ => CNZ);
    end TB_ARCHITECTURE;
```

Здесь типом MICROINST задан формат микрокоманды, подаваемой на входы тестируемого устройства. Поля микрокоманды можно добавлять при необходимости более глубокого тестирования. Константой – массивом mp задано содержимое ПЗУ микропрограммы, которое реализовано в операторе ROM_MP. Это содержимое представляет собой последовательность микрокоманд – входных воздействий для AU. Его можно изменять и дополнять в соответствии с алгоритмом тестирования. Длина микропрограммы задается константой n. В процессе CTM задан счетчик микрокоманд, который выполняет счет по модулю n.

Суть тестирования заключается в подаче данных в FM из полей микрокоманд, выполнение над ними операций в LSM и MPU, записи результатов в FM, чтении их из него и их проверке с контрольными результатами вычислений. Эта проверка может быть выполнена вручную при исследовании временных диаграмм в процессе моделирования. Также можно применить подход, использованный в предыдущей лабораторной работе, т.е. проверять результаты оператором **assert** в определенные моменты времени.

Вопросы по лабораторной работе.

Как выбирается перечень операций AU?

Какие источники операндов в AU?

Как повышают производительность AU?

Как в AU выполняют операции сдвига?

Как в процессорах организованы вызов процедуры и возврат из нее и какую роль в этом играет AU?

Какие признаки запоминают в регистре состояния AU?

Приведите пример процесса, описывающего регистр с мультиплексором на входе.

Как на VHDL запрограммировать ПЗУ микропрограмм?

4.8. Ядро микропроцессора

(Лабораторная работа 7)

Цель лабораторной работы:

овладеть знаниями и практическими навыками по проектированию сложных программноуправляемых блоков, таких как ядро микропроцессора (CPU).

Теоретические сведения

Каждый микропроцессор включает в себя ядро и устройства, подключаемые к нему через его интерфейс, такие как блоки памяти, устройства ввода-вывода, сопроцессоры, расширитель входа прерываний. Ядро микропроцессора описывается его архитектурой с точностью до временных задержек, особенностей элементной базы и т.п. В описание архитектуры микропроцессора входят: система команд, конфигурация памяти, системы адресации, прерываний, защиты памяти, поддержки надежности и тестирования, структура ядра, его интерфейс, основные рабочие режимы и т.п.

От выбора системы команд зависят параметры микропроцессора. Но оптимизация системы команд - это сложный процесс разрешения ряда противоречий. Если выбирать формат команд, в котором команды закодированы компактным образом, то в результате получим программы, которые занимают минимум пространства в памяти программ. Но тогда усложняется схема дешифрации команд и увеличивается ее задержка, что приводит к уменьшению быстродействия процессора.

Обычно для уменьшения длины программ применяют команды различной длины и систему сложных команд. В первом случае минимизация длины программ напоминает метод кодирования Хафмана. Во втором случае одна сложная команда заменяет несколько простых команд. Правда, при этом число возможных команд ограничивается разрядностью поля кода операции и сложностью схемы управления. В обоих случаях усложняются схемы дешифрации и выборки команд, затрудняется повышение быстродействия за счет распараллеливания вычислений.

Архитектура RISC – процессоров основана на том, что они имеют простой, пусть даже логически избыточный формат команд. Это обеспечивает несложную и, следовательно, быструю дешифрацию команд и адресов, возможность выборки и дешифрации нескольких команд одновременно. Существует тенденция к разработке систем команд таких, как у RISC – процессоров. При этом для сохранения программы в компактной форме после ее компиляции она упаковывается специальным компрессором, а перед исполнением – распаковывается аппаратным декомпрессором.

Все современные микропроцессоры исполняют команды в конвейерном режиме. При этом исполнение каждой команды проходит несколько последовательных стадий: выборка команды, ее дешифрация, выборка операндов, операция с операндами, запись результатов. Эти стадии реализу-

ются на различных ступенях конвейера, так что несколько команд выполняются в процессоре одновременно, но на различных стадиях. Поток команд в командном конвейере обрывается на командах переходов. Тогда конвейер освобождается от недообработанных команд, а при выполнении команды по адресу перехода - заполняется снова. Часто такие простои конвейера минимизируют, выполняя команд «задержанного» перехода.

Пример описания CPU

Рассмотрим пример проектирования простого 16- разрядного CPU на базе блоков AU, RAM и ICTR, спроектированных в предыдущих лабораторных работах. В CPU применяется непосредственная адресация (операнд непосредственно в команде) и косвенная адресация одного операнда в основной памяти, а также трехадресная адресация регистровой памяти. Адресное пространство процессора включает в себя 8 регистров FM и 8192 ячейки памяти RAM для хранения 16-разрядных данных и команд, а также 32 регистра периферийных устройств. CPU выполняет набор команд, представленный в таблице 4.8.

Таблица 4.8. Ситема команд ядра микропроцессора

OP	W0							W1
	15 14 13	12 11	10	9 8	7 6	5 4 3	2 1 0	15…0
BRA	0 0 0	COND		DISP				
LJMP	0 0 1	ADDR						
CALL	0 1 0	ADDR						
LD	0 1 1	0 0	–	AQ		AB	–	
SD	0 1 1	0 1	–	–		AB	AD	
IN	0 1 1	1 0	APH	AQ		APL	–	
OUT	0 1 1	1 1	APH	–		APL	AD	
ALOP	1 0 0	ACOP		AQ		AB	AD	
LI	1 0 1	–	–	AQ		–	–	IDATA
RET	1 1 0	–	–	–		–	–	

По команде BRA выполняется условный переход относительно счетчика команд на смещение, определяемое непосредственной константой DISP. Так как эта константа представляет собой 10-разрядное число со знаком в дополнительном коде, то возможен переход на 512 ячеек памяти программ вперед и на столько же – назад. Семантика кода условия перехода COND приведена в табл. 4.9.

По команде LJMP выполняется безусловный (длинный) переход по абсолютному 13- разрядному адресу ADDR. По такому же адресу вызывается подпрограмма в команде CALL. При этом текущий адрес команды, увеличенный на 1, т.е. адрес возврата, должен сохраняться в регистре стека, т.е. в 7-м регистре FM (см. лабораторную работу 6). Противоположная ей команда RET выполняет возврат из подпрограммы и по ней адрес возврата из 7-го регистра переписывается в счетчик команд.

Таблица 4.9. Код условия команды

Мнемоника	COND	Функция от C,N,Z	Описание
NOP	0 0 0	0	Нет операции
JUMP	0 0 1	1	Безусловный переход
NEQ	0 1 0	Not Z	Результат ≠ 0
EQ	0 1 1	Z	Результат = 0
GE	1 0 0	not(C xor N)	Результат ≥ 0
LT	1 0 1	C xor N	Результат < 0
NCY	1 1 0	Not C	Перенос = 0
CY	1 1 1	C	Перенос = 1

По команде LD данное из памяти по адресу, хранящемуся в регистре AB, загружается в регистр AQ. Т.е. здесь выполняется индексная адресация чтения из памяти. Аналогично по команде SD данное из регистра AD Пересылается в память по адресу, хранящемуся в регистре AB. Такая адресация позволяет выполнять доступ к дополнительной внешней памяти данных объемом до 56 Кслов.

Команды IN и OUT предназначены для ввода с периферийного устройства в регистр AQ и соответственно, вывода данного из регистра AD. При этом регистры периферийного устройства могут находиться в старших адресах адресного пространства процессора с младшими разрядами, задаваемыми 5-разрядным кодом AP, состоящим из поля старших разрядов APH и поля младших разрядов APL. Таким образом, процессор может иметь до 32 регистров периферийных устройств.

Команды ALOP выполняют арифметические и логические команды в AU (см. лабораторную работу 6). Их коды F и L образуют управляющее слово для AU, так что ACOP = F & L.

Команда LI загружает в регистр AQ непосредственный операнд, т.е. константу IDATA, которая представлена во втором слове команды. Эта команда играет большую роль при задании адресов записи-чтения данных для команд LD и SD.

В системе команд незадействовано большое количество комбинаций кодов команд. Эти комбинации можно будет использовать в будущем при необходимости расширения системы команд.

Разработанный в лабораторной работе 2 блок ICTR не соответствует данной архитектуре, так как обеспечивает приращение адреса до 4 единиц, в то время как требуется приращение и декремент до 512 единиц. Кроме того, блок должен выдавать адрес возврата при вызове подпрограмм и загружать в счетчик команд сохраненный адрес возврата при возврате из подпрограммы. Также он должен подавать на адресный вход RAM адрес данного при выполнении команд записи и чтения RAM. Структура модернизированного блока ICTR показана на рис.4.13. Так как в ICTR добавлен вход I приращения адреса, вход ARETI загрузки адреса возврата, вход B адреса данного, то изменилась кодировка управляющего слова. Его семантика объясняется в табл. 4.10.

Таблица 4.10. Дешифрация команд перехода

Команда	F	Действие
-	X0XX	Остановка
Не переход	X100	CTR=CTR+1 A=CTR
LJMP, CALL	X101	CTR=D, A=CTR
RET	X110	CTR=ARETI, A=CTR
BRA	X111	CTR=I, A=CTR
LD,SD	1XXX	A=B

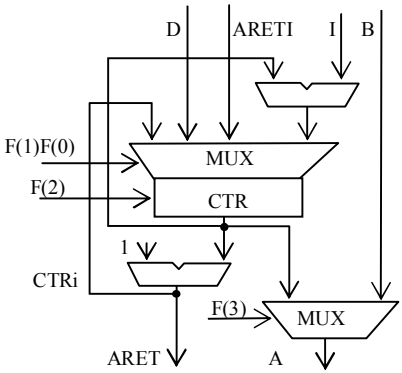


Рис.4.13. Структура ICTR

Архитектура ICTR описывается так:

```
    Library IEEE;
use IEEE.NUMERIC_BIT.all;
entity ICTR2 is port(CLK, RST: in BIT;
-- синхровход и  сброс
                     D : in BIT_VECTOR(12 downto 0);    -- адрес перехода
                     I :  in BIT_VECTOR(9 downto 0);      -- приращение адреса
                     B : in BIT_VECTOR(12 downto 0);     -- адрес данного
                     F : in BIT_VECTOR(3 downto 0);       -- функция
                     ARETI: in BIT_VECTOR(12 downto 0); --адрес возврата
                  A : out BIT_VECTOR(12 downto 0);   -- выходной адрес
                     ARET : out BIT_VECTOR(12 downto 0));-- адрес для сохранения
end ICTR2;
architecture BEH of ICTR2 is
        signal CTR,CTRi:SIGNED(12 downto 0);--счетчик адреса команды
begin
 ICTR:process(RST,CLK,CTR)
        begin
           CTRi<=CTR+1;   -- инкремент адреса
          if RST='1' then
                CTR <="0000000000000";    --сброс регистра счетчика
          elsif CLK='1' and CLK'event then
                case F(2 downto 0) is       -- мультиплексор и регистр адреса
                   when "100"=> CTR<= CTRi;              -- увеличение на 1
                   when "101"=>CTR<= SIGNED(D);    --загр. абс. адреса
                   when "110"=>CTR<= SIGNED(ARETI);--загр. адреса возврата
                   when "111"=>CTR<= CTR+SIGNED(I);-- с приращением
                   when others => null;
                end case;
          end if;
        end process;
 MUX_A:A<=B when F(4)='1' else BIT_VECTOR(CTR);  --выходной мультиплексор
      ARET<=BIT_VECTOR(CTRi);  -- адрес возврата из ПП
    end BEH;
```

Здесь сигналы CTR - регистра – счетчика адреса и CTRi - выхода схемы сложения с 1 – объявлены как число со знаком. Сложение с этими сигналами отображается в сумматоры чисел в дополнительном коде. Поэтому входные сигналы вначале переводятся в подтип SIGNED. И наоборот, выходные адреса формируются как переход типа из SIGNED в BIT_VECTOR.

Структура процессорного ядра показана на рис.4.14. Кроме ICTR, AU, RAM она содержит регистр команд IRG, командный блок управления COP и систему шин передачи данных и управления. Регистр IRG состоит из частей IRG0 и IRG1 для первого и второго слова команды. С регистра IRG0 выдаются адреса регистров AD, AB, AQ, адрес периферийного устройства AP, адрес перехода ADDR и смещение адреса DISP. Через мультиплексор MUXD на AU подается данное с шины DII или непосредственный операнд с IRG1.

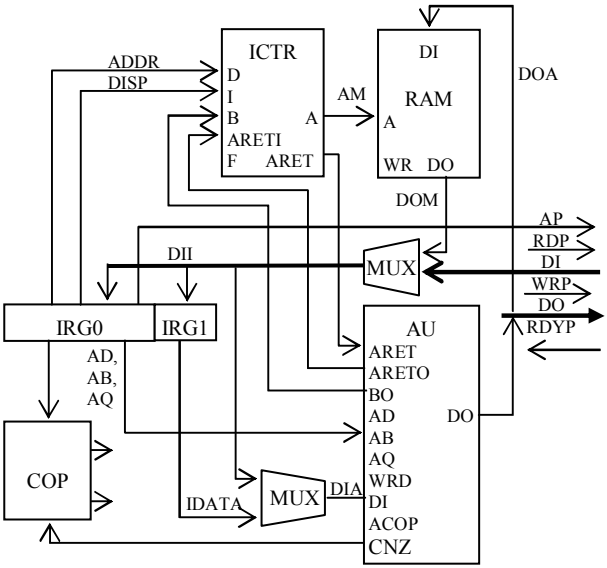


Рис.4.14. Структура ядра микропроцессора

Шины адреса регистра ввода - вывода ADDRP и данных DI, DO с мультиплексором MUXI образуют интерфейс ввода-вывода. При этом в периферийное устройство, выбранное по адресу AP, записывается данное по сигналу WRP и с него считывается данное по сигналу RDP. Если периферийное устройство медленное, то оно выдает сигнал готовности RDYP не сразу, а с присущей задержкой. Блок COP дешифрирует код команды, условие перехода и выдает необходимые управляющие сигналы в нужной последовательности. Его VHDL-описание приведено ниже.

```
entity COP is port(CLK,RST: in BIT;
                   RDYA : in BIT;              -- готовность операции в AU
                   RDYP : in BIT;              -- готовность периферийного у-ва
                   IRG0 : in BIT_VECTOR(15 downto 0);--регистр команды
                   CNZ : in BIT_VECTOR(2 downto 0); -- признаки из AU
```

```
            LINST0 : out BIT;          --записать 1 слово команды
            LINST1 : out BIT;          --записать 2 слово команды
            EIRG : out BIT;            --управление MUXD
            EDI : out BIT;          --управление MUXI
            START : out BIT;        --пуск AU
            RET : out BIT;          --команда RET в AU
            WRRET : out BIT;        --запись адр. возврата в AU
            RD : out BIT;           --чтение AU
            RDP : out BIT;          --чтение периферийного у-ва
            WR : out BIT;           --запись RAM
            WRD : out BIT;          --запись в AU
            WRP : out BIT;          --запись в периферийное у-во
            FI : out BIT_VECTOR(3 downto 0));--управление ICTR
end COP;
architecture BEH of COP is
        type STATE is (norm, ldimm, r_wd, jump); --состояния автомата
        signal st:STATE;                    --регистр состояния автомата
        signal CALL,BRA,LJMP,RETi: BIT;     --команды перехода
        signal LD,SD,LI,\IN\,\OUT\,ALOP : BIT;  -- команды арифм. и пересылок
        signal OP,COND: BIT_VECTOR(2 downto 0);--поля команды
        signal F: BIT_VECTOR(1 downto 0);
        signal condy:BIT;           --логическое условие перехода
        signal del:BIT;             --ожидание
begin
        OP<=IRG0(15 downto 13); --Выделение полей команды
        F<=IRG0(12 downto 11);
        COND<=IRG0(12 downto 10);
--  Дешифратор команды
        BRA <= '1' when OP="000"  else '0';
        LJMP <='1' when OP="001"  else '0';
        CALL <='1' when OP="010"  else '0';
        LD  <=  '1' when OP="011"  and F="00" else '0';  --LD
        SD  <=  '1' when OP="011"  and F="01" else '0';  --SD
        \IN\  <='1' when OP="011"  and F="10" else '0';  --IN
        \OUT\<='1'when OP="011"  and F="11" else '0';  --OUT
        ALOP<='1' when OP="100"  else '0';              --ALOP
        LI   <=  '1' when OP="101"  else '0';           --LI
        RETi <='1' when OP="110"  else '0';
-- Мультиплексор условия перехода ------------------------
        with COND(2 downto 1) select
         condy<= COND(0) when "00",
                 COND(0) xnor CNZ(0) when "01",
                 COND(0) xnor (CNZ(2) xor CNZ(1)) when "10",
                 COND(0) xnor CNZ(2) when "11";
----  Автомат процессора          -------------------------------
        FSM:process(CLK,RST)
        begin
           if RST='1' then
                 st<=norm;
           elsif CLK='1' and CLK'event then
```

```
                case st is
                when norm=> if LI='1' then -- обычное состояние
                                    st<=ldimm;
                           end if;
                           if (LD or SD or ((\OUT\ or \IN\) and RDYP))='1' then
                            st<=r_wd;
                         end if;
                         if (BRA and condy)='1' or (LJMP or CALL or RETi)='1' then
                          st<=jump;
              end if;
        when ldimm=>st<=norm;--последний такт команды LI
        when r_wd=> st<=norm;--последнийтакт ком. LD,SD,IN,OUT
              when jump=> st<=norm;--последний такт ком. BRA,LJMP,CALL,RET
             end case ;
           end if;
        end process;
  ---- Шифратор выходных сигналов ------------------------------
        del<=(ALOP and not rdya)              -- ожидание конца ариф. операции
             or ((\IN\ or \OUT\) and not RDYP); --ожидание  периф. у-ва
        FI(3)<='1' when ((LD or SD)='1') and (st/=r_wd)   else '0';
        FI(2)<='0' when del='1' or (((LD or SD)='1') and (st=norm)) or \OUT\='1'
              or (BRA='1'and st/=jump and condy='0') else'1'; --остановка ICTR
        FI(1 downto 0)<="01" when LJMP='1' or CALL='1' else       --LJMP, CALL
               "10" when RETi='1' else                             --RET
               "11" when BRA='1' and st/=jump and condy='1' else  --BRA
               "00" ;                                              -- +1 ICTR
        LINST0<= '1' when (del='0' and st=norm          --запись 1 слова команды
                    and ((BRA and condy) or CALL or RETi or LJMP or LI
                    or \IN\ or \OUT\ or  LD or SD)='0') or st/=norm else '0';
        LINST1<='1' when LI='1' and st/=ldimm else '0';--запись 2 слова команды
        WRRET<='1' when CALL='1' and  st/=jump else '0';
        WR<='1' when st=norm and SD='1' else '0';
        WRD<='1' when ((LD or \IN\) ='1'and st/=r_wd) -- запись с RAM и периф.
                   or st=ldimm  else '0';                -- непоср. операнда  в FM
        EDI<= '1' when \IN\='1' and st=norm else '0';
        EIRG<='1' when  st=ldimm else '0';
-- Выходные сигналы, которые не зависят от состояния автомата
        START<= ALOP;
        RET<=RETi;
        RD<=SD or \OUT\;
        RDP<=\IN\;
        WRP<=\OUT\;
end BEH;
```

Управляющий автомат FSM имеет 4 состояния: norm - обычное состояние, ldimm –последний такт команды LI, r_wd –последний такт команд LD, SD, IN, OUT и jump – последний такт команд BRA, LJMP, CALL, RET. Его диаграмма состояний показана на рис.4.15.

Все команды выполняются в состоянии процессора norm, в котором результаты команд записываются по соответствующему адресу. Если программа идет по линейному участку, то в каждом такте в процессоре в состоянии norm в регистр команд IRG0 записывается новая команда, а счетчик команд увеличивается на 1. При этом выборка новой команды выполняется одновременно с исполнением текущей команды, т.е. в процессоре реализован двухступенчатый конвейер команд. Если команда требует нескольких тактов для исполнения, как, например, умножение, обращение к периферийному устройству, то процессор остается в состоянии norm, пока не придет сигнал готовности RDY или RDYP, а счетчик команд приостанавливается.

При выполнении команды LI процессор во втором такте команды входит в состояние ldimm, чтобы записать второе слово команды как непосредственный операнд. В конце выполнения команд пересылок данных процессор переходит в состояние r_wd, в котором шина данных и адреса заняты этой операцией, и в этом такте считывание команд и изменение счетчика команд приостанавливаются. Во втором такте команд перехода процессор входит в состояние jump, в котором собственно и выполняется переход. Причем если условие перехода condy не выполняется, то и переход в команде BRA отменяется и процессор не входит в это состояние.

Безусловный переход выполняется за 1 такт и смещение DISP равно разности адресов команды и команды с меткой перехода. При условном переходе, длящемся 2 такта, счетчик команд увеличивается на 1, поэтому смещение должно быть равно разности адресов минус 1. Пример временных диаграмм процессора можно видеть на рис. 4.17. Интерфейс объекта процессорного ядра CPU имеет такое описание.

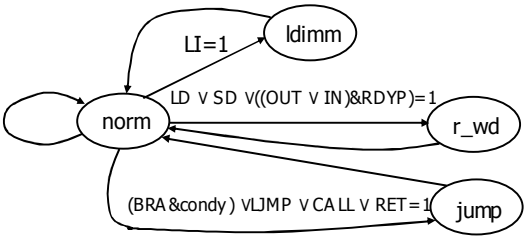


Рис.4.15. Диаграмма состояний FSM

```
entity CPU is port(CLK : in BIT;--Вход синхросигнала
                   RST : in BIT;          --Сброс
                   RDYP : in BIT;         -- готовность периферийного у-ва
                   DI : in BIT_VECTOR(15 downto 0); --входное данное
                   WRP : out BIT;              -- запись периферийного у-ва
                   RDP : out BIT;              -- чтение периферийного у-ва
                   AP : out BIT_VECTOR(4 downto 0); --адрес периф. у-ва
                   DO : out BIT_VECTOR(15 downto 0));--выходное данное
end CPU;
```

Если все блоки процессора описаны в виде объектов, включая мультиплексоры и регистр команды, то архитектуру CPU можно описать с помощью графического редактора системы ActiveHDL. После этого такое описа-

ние автоматически транслируется в VHDL- описание в структурном стиле. Архитектура CPU в графическом виде показана на рис.4.16.

Испытательный стенд для CPU

CPU представляет собой программно управляемое устройство. Для его функционирования необходима программа и данные, хранящиеся в RAM, источник синхросигнала и модель периферийного устройства, которое к нему подключено. Программа одного из возможных испытательных стендов выглядит следующим образом.

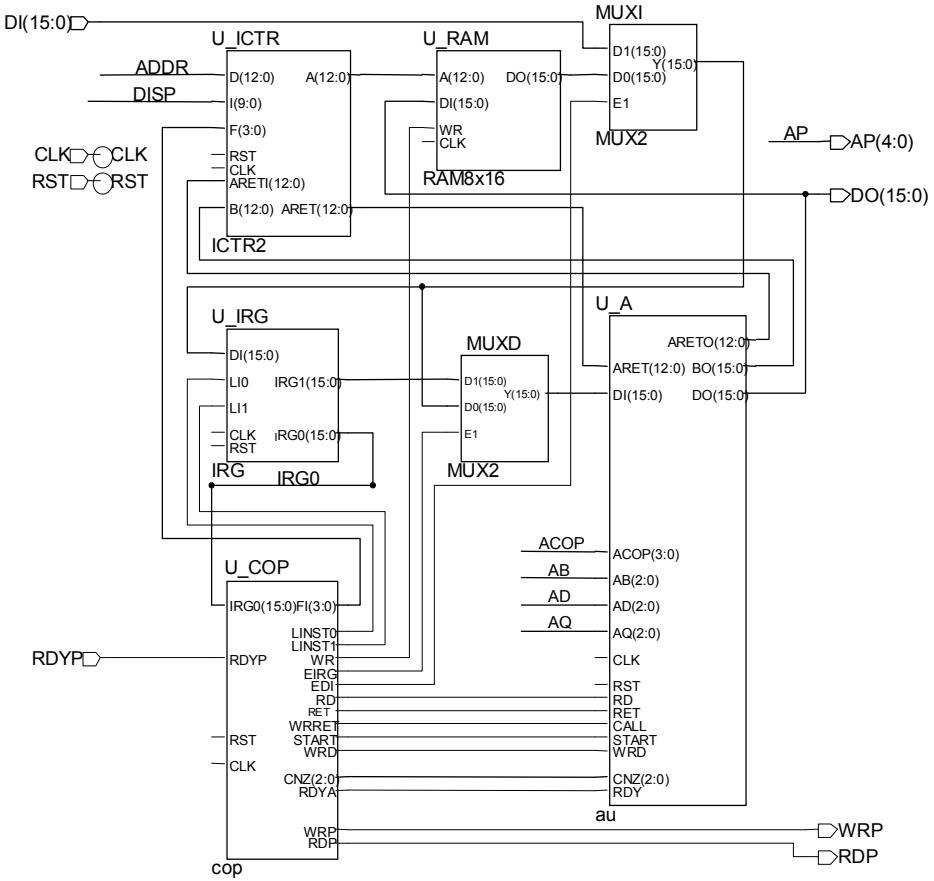


Рис. 4.16. Структура микропроцессорного ядра

```
entity CPU_TB is
end CPU_TB;
architecture TB_ARCHITECTURE of CPU_TB is
        constant TC:time:=10 ns; --перод синхросерии
        component CPU port(CLK,RST : in BIT;
                           RDYP : in BIT;
                           DI : in BIT_VECTOR(15 downto 0);
                           WRP : out BIT;
                           RDP : out BIT;
                           AP : out BIT_VECTOR(4 downto 0);
                           DO : out BIT_VECTOR(15 downto 0) );
        end component;

signal CLK,RST,RDY,selp,WRP,RDP : BIT;
        signal DI, DO,RP1: BIT_VECTOR(15 downto 0);
        signal ADDRP : BIT_VECTOR(4 downto 0);
begin
        CLK<=not CLK after 0.5*TC ;      -- генератор синхросигнала
        RST<='1', '0' after 33 ns;       -- генератор сигнала сброса
        UUT : CPU port map (CLK,RST,     -- тестируемый процессор
                RDYP => RDY,
                DI => DI,
                WRP => WRP,  RDP => RDP,
                AP => ADDRP,DO => DO );
        --------- Модель периферийного устройства        ----------
        selp<='1' when ADDRP="00001" else '0';  --дешифратор адреса
        RDY<= WRP or RDP after TC+3ns;          -- готовность с задержкой
        DI<=X"5A5A" when (selp and RDP) = '1' else --выдача числа
                X"0000" after 5 ns;
        R:process(CLK) begin                    -- Периферийный регистр
                if CLK='1' and CLK'event and selp='1' and WRP='1' then
                        RP1<=DO;
                end if;
        end process;
end TB_ARCHITECTURE;
```

Программа для тестирования должна быть записана в проект RAM как константа начального состояния памяти. Она должна содержать по возможности все команды из системы команд в различном их сочетании. Пример такой программы показан ниже.

Рис.4.17. Структура CPU перед трансляцией в VHDL-описание

```
------- Адреса регист
        constant R0:   BIT_VECTOR(2 downto 0):="000";
        constant R1:   BIT_VECTOR(2 downto 0):="001";
        constant R2:   BIT_VECTOR(2 downto 0):="010";
        constant R3:   BIT_VECTOR(2 downto 0):="011";
        constant R4:   BIT_VECTOR(2 downto 0):="100";
        constant R5:   BIT_VECTOR(2 downto 0):="101";
        constant R6:   BIT_VECTOR(2 downto 0):="110";
        constant R7:   BIT_VECTOR(2 downto 0):="111";
        constant RR0:  BIT_VECTOR(5 downto 0):="000000";
```

```
------- Коды операций  -----------------
      constant BRA:   BIT_VECTOR(2 downto 0):="000";
      constant LJMP:  BIT_VECTOR(2 downto 0):="001";
      constant CALL:  BIT_VECTOR(2 downto 0):="010";
      constant LD:    BIT_VECTOR(6 downto 0):="0110000";
      constant SD:    BIT_VECTOR(6 downto 0):="0110100";
      constant \IN\:  BIT_VECTOR(4 downto 0):="01110";
      constant \OUT\: BIT_VECTOR(4 downto 0):="01111";
      constant ALOP:  BIT_VECTOR(2 downto 0):="100";
      constant LI:    BIT_VECTOR(6 downto 0):="1010000";
      constant RET:   BIT_VECTOR(15 downto 0):=X"C000"; -- "возврат из ПП"
      constant NOOP: BIT_VECTOR(15 downto 0):=X"0000"; -- "нет операции"
------- Функции АЛУ  --------------------
      constant ADD:   BIT_VECTOR(6 downto 0):="1000000";
      constant ADDC:  BIT_VECTOR(6 downto 0):="1000001";
      constant ADDINC:BIT_VECTOR(6 downto 0):="1000010";
      constant SUBDEC:BIT_VECTOR(6 downto 0):="1000100";
      constant SUBC:  BIT_VECTOR(6 downto 0):="1000101";
      constant SUB:   BIT_VECTOR(6 downto 0):="1000110";
      constant \AND\: BIT_VECTOR(6 downto 0):="1001010";
      constant \OR\:  BIT_VECTOR(6 downto 0):="1001110";
      constant \SRL\: BIT_VECTOR(6 downto 0):="1001000";
      constant SRC:   BIT_VECTOR(6 downto 0):="1001001";
      constant \SRA\: BIT_VECTOR(6 downto 0):="1001011";
      constant MUL:   BIT_VECTOR(6 downto 0):="1001100";
------- Условия перехода  ---------------
      constant NOP:   BIT_VECTOR(2 downto 0):="000";
      constant JUMP:  BIT_VECTOR(2 downto 0):="001";
      constant NEQ:   BIT_VECTOR(2 downto 0):="010";
      constant EQ:    BIT_VECTOR(2 downto 0):="011";
      constant GE:    BIT_VECTOR(2 downto 0):="100";
      constant LT:    BIT_VECTOR(2 downto 0):="101";
      constant NCY:   BIT_VECTOR(2 downto 0):="110";
      constant CY:    BIT_VECTOR(2 downto 0):="111";
type MEM8KX16 is array(0 to 8191) of BIT_VECTOR(15 downto 0);


-- Начальное состояние памяти = программа+ данные
constant RAM_init: MEM8KX16:=  -- начальное состояние памяти
-- адрес,    КОП,     операнды,          комментарии
(0=> SUB       &R0&R0&R0,                  -- R0=0 -1я команда
1=>  LI        &R1&RR0,            -- непосредственная константа в R1
2=>  X"0040",                      -- константа –2-е слово команды
3=>  LD        &R2&R1&R0,                  -- данное в R2 из RAM[R1]
4=>  ADDINC  &R1&R1&R0,           -- R1=R1+1
5=>  SUBDEC  &R2&R0&R2,           -- вычли 1: R2=R2-1, т.е. счетчик
6=>  BRA       &NEQ&"1111111101", --переход на адрес-2, если не 0
7=>  LJMP      &"0000000110000", -- длинный переход на адрес 48
8=>  CALL      &"0000000100000",   -- вызов ПП по адресу 32
9=>  \SRL\     &R2&R0&R4,                  -- R2=R4/2
```

```
10=> SD        &R0&R2&R6,     -- по адресу (R2) записывает операнд из R6,
11=> NOOP,
12=> BRA       & JUMP&"1111111111", --вечный цикл окончания программы
-- Подпрограмма -----
-- умножает число R4 на число из периферийного устройства 001 и
-- старшее слово результата записывает в периферийное устройство 001
32=> \IN\      &"00"&R3&"001"&R0,  --  ввод данного
33=> MUL       &R5&R4&R3,                  -- R5,R6=R4*R3
34=> \OUT\     &"00"&R0&"001"&R5, -- вывод данного
35=> RET,
--   Отработка длинного перехода
48=> ADDINC   &R4&R18&R1,          -- R4=R1+R1
49=> LJMP      &"0000000001000",-- переход на адрес 8
-- Область данных
64=> X"0004",                      -- исходное данное
      others=> X"0000");
```

Для того чтобы составление программы приближалось к программированию на ассемблере, вначале были объявлены константы, имена и длина которых совпадают с мнемоникой и длиной соответствующих полей команды. Тогда слово команды представляет собой конкатенацию таких имен констант и необходимых литералов.

В программе из памяти по адресу 64 считывается данное 4 - начальное значение счетчика. После выполнения цикла декремента этого счетчика адрес увеличивается до 68. После длинного перехода и возврата обратно это число удваивается до 136. Наконец, после умножения его в подпрограмме на число 5A5Ah=23130 получается произведение 2FFFD0h, старшая часть которого записывается в периферийный регистр, а младшая – в RAM. И в конце программа зацикливается на адресах 11,12. Таким образом, в этой тестовой программе исполняются все основные команды, и при успешном ее выполении в регистре периферийного устройства будет код 2Fh, а в памяти по адресу 68 – код FFD0h.

Временные диаграммы работы процессора после его сброса при выполнении тестовой программы показаны на рис.4.17.

Таким образом, разработан CPU, выполняющий все арифметико-логические команды, кроме умножения за 1 такт, а остальные команды – в основном, за 2 такта. После синтеза и конфигурирования CPU в ПЛИС Virtex он занимает до 370 ЭКЛБ и работает с тактовой частотой до 50 МГц, т.е. он выполняет около 40 млн. операций в секунду.

Вопросы по лабораторной работе.

Какие особенности испытательного стенда для CPU?
Что такое архитектура процессора и какие ее признаки?
Как оптимизируют систему команд CPU?
Каковы архитектурные отличия RISC - процессоров?
В чем заключается конвейерная работа CPU?
Какие бывают типы адресации CPU?
Чем отличается абсолютный переход от относительного перехода?

Как в CPU выполняются вызов подпрограммы и возврат из нее?
Объясните действие общей шины на базе мультиплексоров.
В чем состоит особенность мультиплексора условия перехода?
Как на VHDL проектируют канонический автомат Мили?

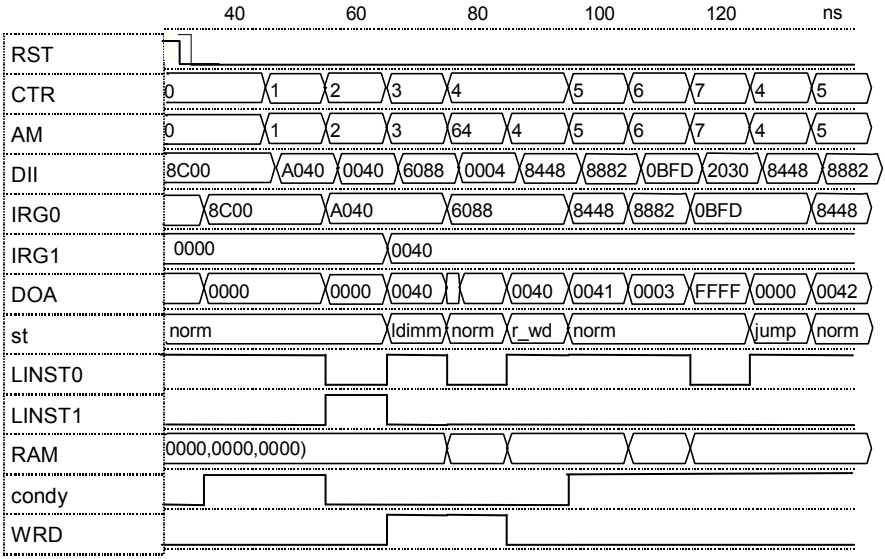


Рис.4.17. Временные диаграммы процессора при исполнении тестовой программы

ЛИТЕРАТУРА

1. Сергиенко А.М. Особенности VHDL как языка параллельного программмирования// Электрон. Моделирование. -Т.25.-2003. -№3. –С.115-123.

2. Шкурко А.И. и др. Компьютерная схемотехника в примерах и задачах. -К.: "Корнійчук". – 2003. –144с.

3. Сергиенко А.М. VHDL для проектирования вычислительных устройств. –К.:"Корнійчук","ДиаСофт". –2003. –203 с.

4. http://www.aldec.com.ua

5. http://www.xilinx.com

ПРИЛОЖЕНИЕ

```
------------------------------------------------------------------------------------------------
--
-- Title      : CNetlist_Lib
-- Design     : Computer Network Engineering
-- Author     : Anatolij Sergyienko
-- Company    : NTUU "KPI"
------------------------------------------------------------------------------------------------
-- File       : CNetwork_Lib.vhd
-- Generated  : Thu Sep 23 08:55:32 2004
------------------------------------------------------------------------------------------------
-- Description : Библиотека функций и объектов
--             для разработки логических схем на основе ПЛМ и ПЛИС
--             основные типы - integer, bit и bit_vector
------------------------------------------------------------------------------------------------
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
Package CNetwork is
        constant MINUS1: bit_vector(31 downto 0):=(31=>'1', others=>'0');
        subtype Z01 is STD_uLOGIC range '0'to'Z';-- '0' to 'Z';
        --преобразование вектора бит - числа в прямом коде в целое
        function BIT_TO_INT(b:bit_vector) return integer;

        --преобразование вектора бит - числа в доп.коде в целое
        function BITS_TO_INT(b:bit_vector) return integer;

        --преобразование бита в целое
        function BIT_TO_INT(b:bit) return integer;

        --преобразование целого со знаком в вектор бит - число в прямом коде
        function INT_TO_BIT(i,l:integer) return bit_vector;

        --преобразование целого со знаком в вектор бит - число в доп.коде
        function INT_TO_BITS(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 CNetwork;
```

```
Package body CNetwork is

        function BIT_TO_INT(b:bit_vector) return integer is
                variable bi:bit_vector(b'range);
                variable fl:boolean:=false;
                variable t,j:integer:=0;
        begin
        assert b'length<=32 report "слишком длинный вектор бит" severity failure;
                --          переводим в целое
                for i in b'reverse_range loop
                        if b(i)='1' then
                                t:=t+2**j;
                        end if;
                        j:=j+1;
                end loop;
                return t;
        end function;

        function BITS_TO_INT(b:bit_vector) return integer is
                variable bi:bit_vector(b'range);
                variable fl:boolean:=false;
                variable t,j:integer:=0;
        begin
        assert b'length<=32   report "слишком длинный вектор бит" severity failure;
                bi:=b;
                --получаем прямой код
                if b(b'left)='1' then
                        for i in b'reverse_range loop
                                if not fl and b(i)='1' then
                                        fl:=true;
                                elsif fl then
                                        bi(i):=not b(i);
                                end if;
                        end loop;
                end if;
                -- и         переводим в целое
                for i in b'reverse_range loop
                        if bi(i)='1' then
                                t:=t+2**j;
                        end if;
                        j:=j+1;
                end loop;
                --и представляем со знаком
                if b(b'left)='1' then
                        t:=-t;
                end if;
                return t;
        end function;
```

```
function BIT_TO_INT(b:bit) return integer is
begin
        if b='0' then
                return 0;
        else
                return 1;
        end if;
end function;

function INT_TO_BIT(i,l:integer) return bit_vector is
        variable bv: bit_vector(l-1 downto 0):=(others=>'0');
        variable ii,i2:integer;
        variable fl:boolean:=false;
begin
        assert l<=32 report "слишком длинный вектор";
        ii:=i;
        --построение вектора битов
        for j in bv'reverse_range loop
                i2:=ii/2;
                if i2*2/=ii then
                        bv(j):='1';
                end if;
                ii:=i2;
        end loop;
        return bv;
end function;

function INT_TO_BITS(i,l:integer) return bit_vector is
        variable bv: bit_vector(l-1 downto 0):=(others=>'0');
        variable ii,i2:integer;
        variable fl:boolean:=false;
begin
        assert l<=32 report "слишком длинный вектор";
        ii:=abs(i);
        --построение вектора битов
        for j in bv'reverse_range loop
                i2:=ii/2;
                if i2*2/=ii then
                        bv(j):='1';
                end if;
                ii:=i2;
        end loop;
        --получение доп.кода
        if i<0 then
                for j in bv'reverse_range loop
                        if not fl and bv(j)='1' then
                                fl:=true;
                        elsif fl then
                                bv(j):=not bv(j);
                        end if;
```

```
                        end loop;
                end if;
                return bv;
        end function;

        function INT_TO_BIT(i:integer) return bit is
        begin
assert i=0 or i=1 or i=integer'left report"неизвестной длины вектор" severity failure;
                if i=0 then
                        return '0';
                else
                        return '1';
                end if;
        end function;


        function LOG_TAB(e,d,c,b,a:BIT:='0';mask:bit_vector) return bit is
                variable adr: integer:=0;
                variable v:bit_vector(4 downto 0);
        begin
                v:=e&d&c&b&a;
                for i in 0 to 4 loop
                        if v(i)='1' then
                                adr:=adr+2**i ;
                        end if;
                end loop;
                return mask(adr);
        end function ;

end package body;


use Cnetwork.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;

architecture BEH of LUT4 is
begin
        y<=LOG_TAB('0',d,c,b,a,mask) after td;
end BEH;
use Cnetwork.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;

architecture BEH of LUT5 is
begin
        y<=LOG_TAB(e,d,c,b,a,mask) after td;
end BEH;

-- генератор случайных чисел с равномерным распределением
Library IEEE;
Use IEEE.MATH_REAL.ALL;
use Cnetwork.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;

architecture BEH of RANDOM_GEN is
begin
        process
                variable clki:bit;
                variable a,b:positive:=SEED;
                variable s:real;
        begin
                Uniform(a,b,s);
                clki:=not clki;
                CLK<=clki;
                wait for tp/2;
                clki:=not clki;
                CLK<=clki;
                Y<=INT_TO_BIT(integer(s*real(2**n)),n) ;
                wait for tp/2 ;
                end process;
end BEH;
```

```
--Триггер с разрешением записи CE и асинхронным сбросом R
entity FDRE  is generic(td:time:=1 ns);
        port (Q:out  bit;
                D :     in   bit;
                C :     in   bit;
                CE:     in   bit;
                R :     in   bit);
end FDRE;
architecture BEH of FDRE is
        signal qi:bit;
begin
        process(C,R) begin
                if R='1' then
                        qi<='0';
                elsif C='1' and C'event then
                        if CE='1' then
                                qi<=D;
                        end if;
                end if;
        end process;
        Q<=qi after td;
end BEH;

use  Cnetwork.all;
entity PLM_6 is
        generic(td:time:=1 ns);                         -- задержка
        port(A,    B,C,D,E,F: in BIT;
                Y : out BIT);
end PLM_6;
use  Cnetwork.all;
entity PLM_5 is
        generic(td:time:=1 ns);                         -- задержка
        port(A,    B,C,D,E: in BIT;
                Y : out BIT);
end PLM_5;

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

use  Cnetwork.all;
entity PLM_3 is
        generic(td:time:=1 ns);                         -- задержка
        port(A,    B,C: in BIT;
                Y : out BIT);
end PLM_3;
```

Учебное издание

Сергиенко Анатолий Михайлович
Корнейчук Виктор Иванович

Микропроцессорные устройства на программируемых логических ИС

Редактор Клименко М.К.

Видавництво "Корнійчук", 04116, Київ-116, а/с 4
Свідотство про внесення до Державного реєстру суб'єктів видавничої справи №424 від 18.04.2001

Подп. к печати	Формат 60x84 1/16
Бумага газетная	Способ печати офсетній
Усл. печ. лист.	Усл. кр. – отт
Уч.-изд.л.	
Тираж	Заказ №

Фірма „Віпол”
04151, Київ, вул. Волинська, 60