Московский государственный технический университет
имени Н.Э. Баумана

А.Ю. Попов

ПРОЕКТИРОВАНИЕ ЦИФРОВЫХ УСТРОЙСТВ С ИСПОЛЬЗОВАНИЕМ ПЛИС

*Допущено Учебно-методическим объединением вузов
по университетскому политехническому образованию
в качестве учебного пособия для студентов высших
учебных заведений, обучающихся по направлению
230100 «Информатика и вычислительная техника»*

М о с к в а
Издательство МГТУ им. Н.Э. Баумана
2 0 0 9

УДК 681.3(075.8)
ББК 32.973.2
П58

Р е ц е н з е н т ы:
заведующий кафедрой «Вычислительные машины, системы и сети» МАИ, д-р техн. наук, профессор *О.М. Брехов*

начальник отдела НИИ ИСУ МГТУ им. Н.Э. Баумана, канд. техн. наук *А.С. Романовский*

Попов А.Ю.

П58 Проектирование цифровых устройств с использованием ПЛИС: Учеб. пособие. — М.: Изд-во МГТУ им. Н.Э. Баумана, 2009. — 80 с.

ISBN 978-5-7038-3317-9

Рассмотрены вопросы, связанные с проектированием цифровых устройств на основе программируемых логических интегральных схем (ПЛИС). Даны сведения об архитектуре современных серий ПЛИС и области их применения. Приведены основы языка описания аппаратных средств VHDL и примеры описания на нем узлов ЭВМ: ОЗУ, цифровых автоматов, сумматоров, регистров, мультиплексоров, дешифраторов и т. д. Описан маршрут проектирования с использованием САПР Xilinx ISE. Рассмотрен состав этой системы. Описаны ее назначение, интерфейс и способы использования входящих в нее модулей.

Для студентов, обучающихся по направлению «Информатика и вычислительная техника».

УДК 681.3(075.8)
ББК 32.973.2

Учебное издание

Попов Алексей Юрьевич

ПРОЕКТИРОВАНИЕ ЦИФРОВЫХ УСТРОЙСТВ
С ИСПОЛЬЗОВАНИЕМ ПЛИС

Редактор *Е.К. Кошелева*
Компьютерная верстка *С.А. Серебряковой*

Подписано в печать 28.04.2009. Формат 60×84/16.
Усл. печ. л. 4,65. Тираж 300 экз. Заказ .

Издательство МГТУ им. Н.Э. Баумана.
Типография МГТУ им. Н.Э. Баумана.
105005, Москва, 2-я Бауманская ул., 5.

ISBN 978-5-7038-3317-9

© Попов А.Ю., 2009
© МГТУ им. Н.Э. Баумана, 2009

ПРЕДИСЛОВИЕ

Разработка цифровых устройств, обладающих высоким быстродействием и сложностью, невозможна без применения специализированных систем автоматизированного проектирования (САПР), систем моделирования и макетирования. Комплексные требования к готовому изделию, такие как низкая стоимость, малое время разработки и модернизации, предельное быстродействие, заставляют разработчиков использовать универсальные средства и методы проектирования. Указанным качествам в настоящее время соответствуют технология проектирования устройств на основе микропроцессоров, а также технология проектирования с использованием программируемых логических интегральных схем (ПЛИС). В пособии рассматриваются архитектура современных ПЛИС и процесс проектирования устройств на их основе.

1. АРХИТЕКТУРА ПЛИС

1.1. Программируемые логические матрицы и программируемая матричная логика

История развития интегральных схем (ИС) с программируемой структурой насчитывает уже более 30 лет. Основой для современных ПЛИС послужили несколько типов полузаказных ИС, таких как программируемые логические матрицы (ПЛМ, Programmable Logic Array, PLA), программируемая матричная логика (ПМЛ, Programmable Array Logic, PAL), базовые матричные кристаллы (БМК, Gate Array, GA). Появление электронных схем с программируемой логикой функционирования было вызвано потребностью в нестандартных компонентах. Выпуск таких компонентов в виде заказных ИС в большинстве случаев экономически не целесообразен.

ПЛМ позволяют реализовать n логических функций от m аргументов и содержат последовательно соединенные k связями (термами) матрицы элементов И и элементов ИЛИ (рис. 1). Помимо этого в матрице И могут быть использованы инвертированные входные сигналы, что позволяет реализовывать функции в дизъюнктивной нормальной форме (ДНФ).

Изготовленная на заводе ПЛМ содержит матрицы со всеми возможными связями (на рис. 1 показаны точками): матрица И позволяет получить k конъюнкций входных сигналов, а матрица ИЛИ — n дизъюнкций термов. В этом случае программирование заключается в разрушении излишних связей (например, для ПЛМ К556РТ1). В другом варианте в исходной ПЛМ все связи отсутствуют, а программирование заключается в их создании.

Структура ИС ПМЛ позволяет более полно использовать ресурсы кристалла при проектировании простых устройств. В отличие от ПЛМ в таких микросхемах программирование возможно

только для матрицы И, а матрица ИЛИ фиксирована (рис. 2). Ограничения на состав и число термов позволяют усложнить остальные части ПМЛ.

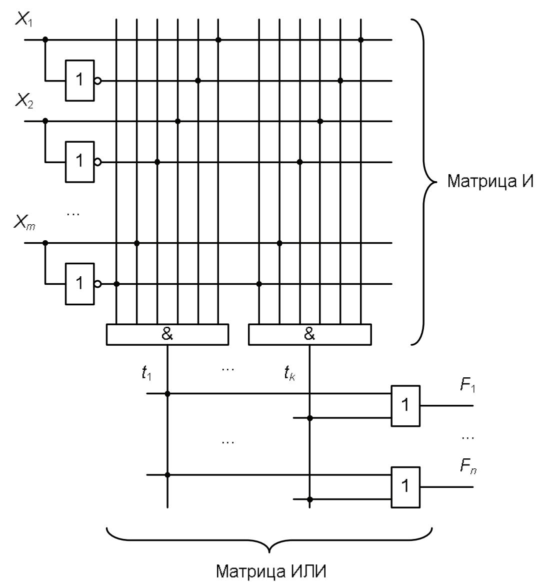

Рис. 1. Структура ПЛМ

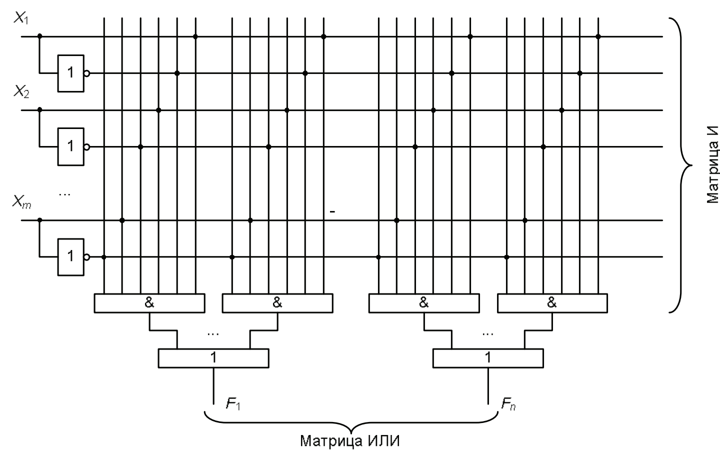

Рис. 2. Структура ПМЛ

С момента своего создания функциональные возможности ПЛМ и ПМЛ были расширены благодаря следующим усовершенствованиям:

• введению двунаправленных, обратных и межэлементных связей, что позволяет наращивать число термов функций (например, отечественные ПЛМ К1556ХП8);

• введению элементов памяти, что позволяет проектировать на основе ПЛМ и ПМЛ синхронные цифровые автоматы;

• программированию выходных буферов для выдачи выходных сигналов в прямом или инверсном виде;

• использованию мультиплексоров для выбора альтернативных путей прохождения сигналов, репрограммируемых точек связи и памяти конфигурации, позволяющих разработчикам неоднократно программировать функциональность и связность частей ПЛМ и ПМЛ.

Результатом эволюции ИС указанных типов стали сложные программируемые логические устройства (СПЛУ, Complex Programmable Logic Devices, CPLD), которые будут подробно рассмотрены ниже.

1.2. Базовые матричные кристаллы

Подобно ПЛМ и ПМЛ, БМК позволяют с малыми экономическими затратами реализовать нестандартную схемную логику. Созданная по этой технологии ИС называется матричной большой интегральной схемой (МаБИС).

Использование БМК основано на том факте, что любое сложное устройство состоит из стандартных функциональных частей, таких как логические или схемотехнические элементы, буферы, усилители и т. д. Для обеспечения требуемой функциональности предусматривается включение в структуру БМК избыточного числа таких частей без их коммутации. Окончательная же функциональность определяется на заключительных этапах производства МаБИС с помощью создания нескольких слоев коммутации, наносимых на стандартную заготовку. Таким образом, заказчик микросхемы с нестандартной логикой индивидуально оплачивает только часть дорогостоящей подготовки к мелко- или среднесерийному производству, в то время как бóльшая часть издержек на

изготовление масок делится на всех заказчиков. Такой подход нашел широкое применение с середины 1970-х годов. В нашей стране выпускались БМК семейств 1806ХМ1, 1515ХМ1, 1593ХМ1, 1537ХМ1, 1592ХМ1 и др. Обобщенная структура БМК показана на рис. 3.

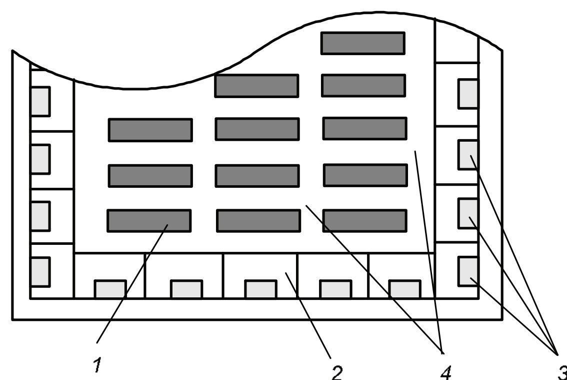

Рис. 3. Структура БМК:
1 — макроячейка; *2* — буферная ячейка; *3* — внешние контактные площадки; *4* — участки для прокладки трасс соединений

Заготовка БМК состоит из макроячеек, содержащих регулярно повторяющиеся базовые ячейки. Состав и взаимное расположение базовых ячеек определяются возможностью максимально эффективной реализации стандартных компонентов (функциональных ячеек). Между макроячейками имеются свободные участки, необходимые для прокладки трасс межсоединений. На периферии БМК размещаются буферные ячейки, содержащие усилители, шинные формирователи и т. д. В непосредственной близости от буферных ячеек располагаются контактные площадки для электрического соединения внутренних цепей с выводами микросхемы.

При совершенствовании функциональных возможностей БМК были использованы следующие принципы:

• наращивание числа базовых ячеек в макроячейках и числа макроячеек на БМК;

• усложнение блоков ввода/вывода (добавление буферов, шинных формирователей, введение двунаправленных выводов и выводов с третьим состоянием);

• введение в структуру БМК законченных функциональных блоков (ОЗУ, ПЗУ и т. д.);

• применение программируемых точек связи совместно с реализацией матриц коммутации и памяти конфигурации, что позволяет перепрограммировать функциональность и связность частей БМК.

Совершенствование технологии изготовления БМК привело к появлению программируемых пользователем вентильных матриц (ППВМ, Field Programmable Gate Arrays, FPGA). Однако из-за невысокой стоимости и достаточно высокого быстродействия производство и применение БМК, ПЛМ и ПМЛ не утратило актуальности. В англоязычной терминологии современные полузаказные ИС, подобные БМК, носят название Mask Programmable Gate Arrays. Используются и другие разновидности полузаказных ИС, таких как ИС на стандартных ячейках (Structured Application Specific Integrated Circuits, Structured ASIC). Наибольшее быстродействие обеспечивают полностью заказные ИС (Application Specific Integrated Circuits, ASIC).

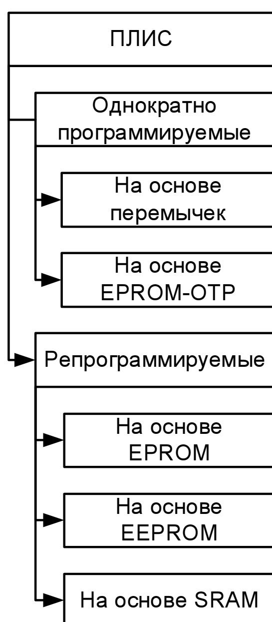

Рис. 4. Классификация ПЛИС по типу программируемых связей

Программирование связности частей современных ПЛИС осуществляется с помощью многочисленных линий, проложенных на кристалле в ортогональных направлениях и соединенных программируемыми точками связи (рис. 4). Однократно программируемые ПЛИС выпускаются с программируемыми точками связи на основе пережигаемых перемычек antifuse, ячеек памяти EPROM-OTP. Такие устройства обладают многими достоинствами: низкой стоимостью, повышенной устойчивостью к воздействию радиации, быстрой готовностью при включении, возможностью обеспечения повышенной секретности проектов и др.

Среди ПЛИС с репрограммируемыми связями наиболее широко распространены устройства на основе триггерной памяти конфигурации (рис. 5). В

ПЛИС других типов используются элементы на ПЗУ с ультрафиолетовым стиранием (EPROM) и репрограммируемым ПЗУ с электрическим стиранием (EEPROM).

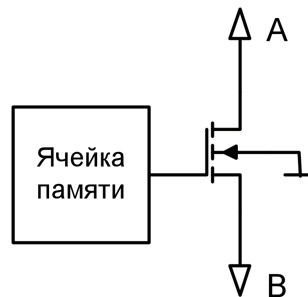

Рис. 5. Программируемая точка связи на основе статической памяти

1.3. ПЛИС типа CPLD

Микросхемы типа CPLD выпускаются многими зарубежными производителями. Среди разработчиков радиоэлектронной аппаратуры наиболее популярны ПЛИС фирм Xilinx, Altera, Actel. Рассмотрим более подробно архитектуру ПЛИС типа CPLD серии XC9500, выпускаемых фирмой Xilinx.

В ПЛИС типа CPLD многократно реализованы элементы ПЛМ или ПМЛ, функциональность которых расширена благодаря добавлению обратных связей, использованию триггерной памяти, применению реконфигурируемых связей и программируемой функциональности. ПЛИС типа CPLD состоит из нескольких простых функциональных блоков (SPLD), соединенных матрицей межсоединений (рис. 6). Серии микросхем могут отличаться друг от друга числом функциональных блоков и их сложностью.

Сигналы, поступающие на входы ПЛИС типа CPLD, с помощью блока ввода/вывода передаются в программируемую матрицу соединений. С помощью этой матрицы также могут быть распределены по кристаллу выходные сигналы блоков SPLD. Глобальные сигналы синхронизации (GCK), сброса и установки (GSR) и управления третьим состоянием (GTS) подводятся ко всем SPLD. Как правило, современные ПЛИС обеспечивают возможность использования нескольких внешних источников синхросигналов, причем различные части устройства могут работать на разных час-

тотах. Часть блоков SPLD, для синхронизации которых используется один и тот же глобальный тактовый сигнал, называется *тактовым доменом*.

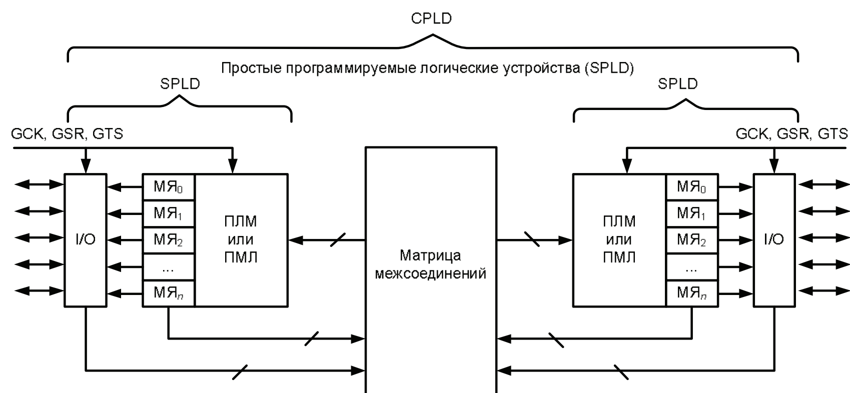

Рис. 6. Структура ПЛИС типа CPLD

Каждый функциональный блок представляет собой матрицу конъюнкторов, обеспечивающих получение термов. Для увеличения функциональности некоторые термы могут быть направлены не в одну, а в несколько соседних макроячеек (рис. 7). В состав макроячейки входят распределитель термов и схема сложения по модулю 2, триггер с динамическим управлением и мультиплексоры *2 – 5* выбора сигналов управления триггером (адресные входы на рисунке не показаны). Входные сигналы матрицы И подаются из матрицы коммутации. В CPLD XC9500, например, таких линий 54. Каждый сигнал подается на матрицу в прямом и инверсном виде. Распределитель термов служит для расширения функциональности макроячеек за счет передачи термов на ближайшие макроячейки.

Полученные из матрицы конъюнкторов и из соседних макроячеек термы передаются на элемент ИЛИ, а также на мультиплексоры управления. Результат дизъюнкции матрицы ИЛИ может быть инвертирован с помощью схемы сложения по модулю 2, если на мультиплексоре *1* выбран сигнал низкого уровня. При выборе на мультиплексоре *1* сигнала высокого уровня сигнал со схемы ИЛИ повторяется. Уровнем этого сигнала также возможно управлять с помощью одного из выходов распределителя термов. Полу-

ченный результат может быть выдан на триггер с динамическим управлением или же непосредственно передан в блок ввода/вывода или в программируемую матрицу соединений с помощью мультиплексора *6*. Для управления выходным буфером, на который подается этот сигнал, в распределителе термов предусмотрен один выход для сигнала управления третьим состоянием.

Асинхронный сброс и установка триггера макроячейки осуществляются с помощью сигнала, выбираемого на мультиплексорах *2* и *3*. Таковыми могут быть как глобальные сигналы сброса (GR) и установки (GS), так и сигнал из распределителя термов. В качестве сигнала синхронизации с помощью мультиплексора *2* может быть использован один из выходов распределителя термов или же сигнал глобальной синхронизации GCK. Мультиплексор *5* позволяет изменить фазу синхросигнала на 180°.

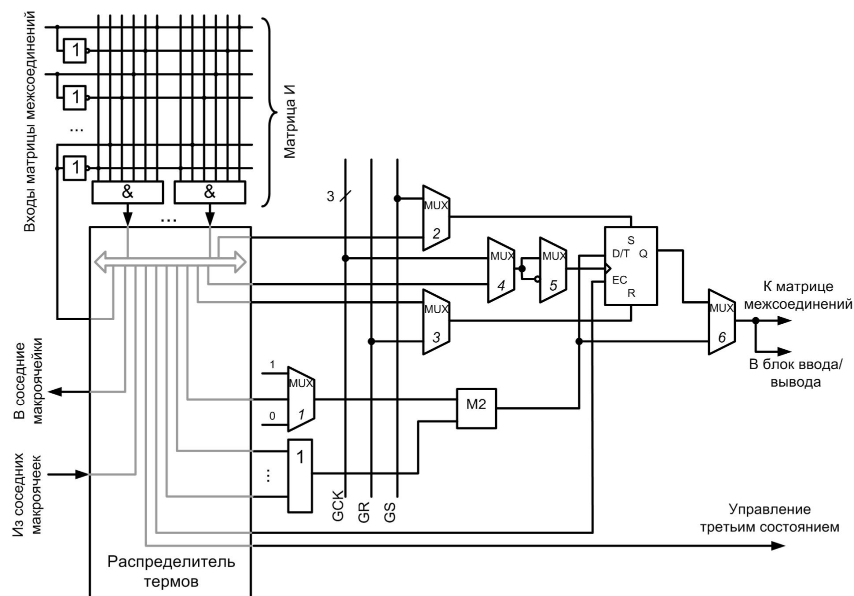

Рис. 7. Структура макроячейки (на примере CPLD XC9500)

Периферийная часть ПЛИС типа CPLD состоит из однотипных элементов — блоков ввода/вывода, предназначенных для соединения внутренних цепей ПЛИС с внешними цепями. Структура блока

ввода вывода основана на шинном формирователе (элементы *1* и *2* на рис. 8), дополненном схемами для подтягивания потенциала и подключения общей точки, мультиплексором управления третьим состоянием (элемент *3*) и схемой управления крутизной фронта.

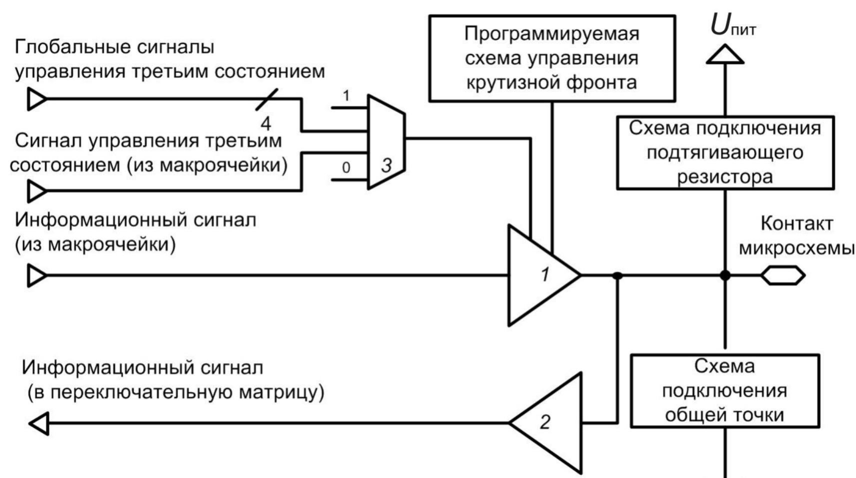

Рис. 8. Структура блока ввода/вывода

ПЛИС типа CPLD выгодно отличаются от ПЛИС других типов простотой и регулярностью структуры. В качестве памяти конфигурации чаще всего используются ячейки типа EEPROM, что повышает секретность проектов. Основным назначением ПЛИС типа CPLD является реализация интерфейсной логики (USB, PCI, PCI-X и т. д.).

1.4. ПЛИС типа FPGA

ПЛИС типа FPGA предназначены для реализации сложных проектов, и их емкость достигает десятков миллионов «эквивалентных вентилей». (Данный термин означает, что для реализации одного и того же проекта на полностью заказной ИС потребовалось бы определенное число вентилей, называемое при реализации проекта с использованием ИС «числом эквивалентных вентилей».) Такая высокая функциональность позволяет объединять на одном

кристалле несколько процессорных устройств и интерфейсную логику. Архитектура FPGA будет рассмотрена на примере ПЛИС распространенной серии Spartan-3 фирмы Xilinx.

Заданную при программировании функциональность обеспечивают конфигурируемые логические блоки (КЛБ), расположенные по всей площади кристалла. Их число является основным параметром, характеризующим возможность реализации на конкретной ПЛИС сложных проектов. Связность КЛБ обеспечивается с помощью переключательной матрицы (рис. 9), состоящей из цепей различной длины: от длинных глобальных до коротких прямых, соединяющих соседние КЛБ. Коммутация цепей переключательной матрицы осуществляется с помощью коммутационных блоков.

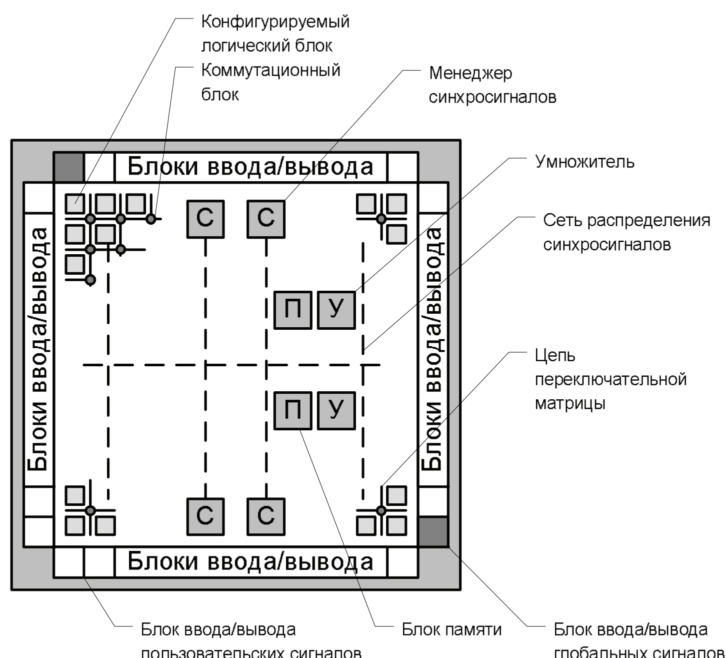

Рис. 9. Структура FPGA

При реализации на основе ПЛИС сложных устройств, работающих на высокой тактовой частоте, особые требования предъявляются к глобальным сигналам синхронизации. В современных

ПЛИС типа FPGA система синхронизации состоит из специально спроектированной сети распределения синхросигналов, основным свойством которой является одинаковая длина линий от источника синхросигнала (от специально выделенного для этих целей контакта ПЛИС или от менеджера синхросигналов) до всех КЛБ.

Так же как и в CPLD, в FPGA предусмотрена возможность использования нескольких тактовых сигналов. Для реализации функций деления, умножения и сдвига по фазе синхросигнала на кристалле предусмотрено несколько менеджеров синхросигналов. Использование обычных цепей передачи информационных сигналов для передачи синхросигналов не рекомендуется. На кристалле также располагаются дополнительные устройства — блоки статической памяти и умножители. На периферии кристалла находятся блоки ввода/вывода.

КЛБ содержат ресурсы, обеспечивающие реализацию комбинационной логики и элементов памяти. Для схем обоих типов в наиболее распространенных ПЛИС со статической памятью конфигурации используются небольшие блоки адресной памяти с произвольным доступом, называемые *таблицами соответствия* (Look Up Table, LUT). Такая память содержит от 16 (реализация 16×1) до 64 (64×1) бит, записываемых в LUT при конфигурации или в рабочем режиме. Также предусматривается возможность использования LUT в качестве сдвигового регистра. Все это существенно расширяет функциональность ПЛИС, так как позволяют реализовать LUT в нескольких качествах:

• при использовании комбинационных схем в качестве таблицы истинности произвольной функции от переменных, подаваемых на адресные входы LUT;

• при реализации автоматов и памяти в качестве блока статической адресной памяти (16×1 или 64×1 бит);

• при реализации арифметических операций, последовательных интерфейсов и конвейеров в качестве регистра сдвига.

Например, КЛБ в FPGA Spartan-3 (рис. 10) содержит четыре секции, две из которых могут функционировать как память с произвольным доступом (RAM) или как регистр сдвига (SRL в блоке SLICEM). Две оставшиеся секции могут функционировать только в качестве постоянных запоминающих устройств (ROM).

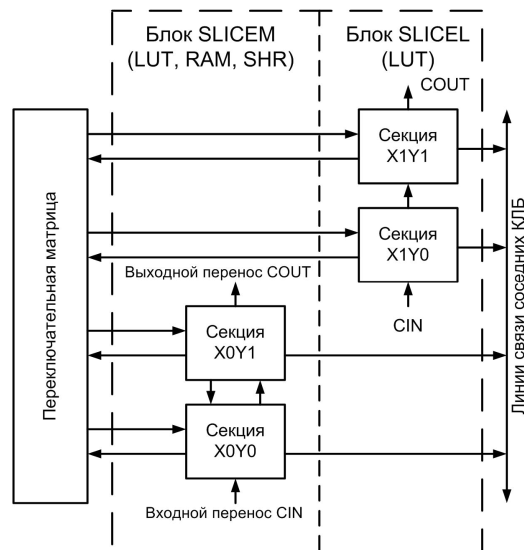

Рис. 10. Структура КЛБ (на примере Spartan-3)

Каждая секция в блоке типа SLICEL (рис. 11) состоит из двух блоков LUT, двух схем сложения по модулю 2, схем формирования ускоренного переноса (элементы *1*, *2*, *3*, *7*, *8*, *9*, *13*), мультиплексоров выбора режима (элементы *5*, *6*, *11*, *12*), двух динамических триггеров, схемы наращивания размерности мультиплексора (элемент *10*), независимого мультиплексора *4*. Адресные входы мультиплексоров на рис. 11 не показаны. Блок SLICEM отличается от блока SLICEL тем, что в нем реализована дополнительная схема управления сдвигом.

Секция в блоке SLICEL может функционировать в качестве:

• комбинационной схемы, реализующей функции Y и X от четырех переменных (G1..G4 и F1..F4) с помощью блока LUT и мультиплексоров *5* и *11*;

• мультиплексора *4* с двумя входами FXINA и FXINB и адресным входом BY;

• комбинационной схемы, реализующей функцию F5 от девяти переменных (G1..G4, F1..F4, BX) на блоке LUT и мультиплексоре *10*;

• мультиплексора с пятью адресными линиями (G1..G4 или F1..F4, BX) на блоке LUT и мультиплексоре *10*;

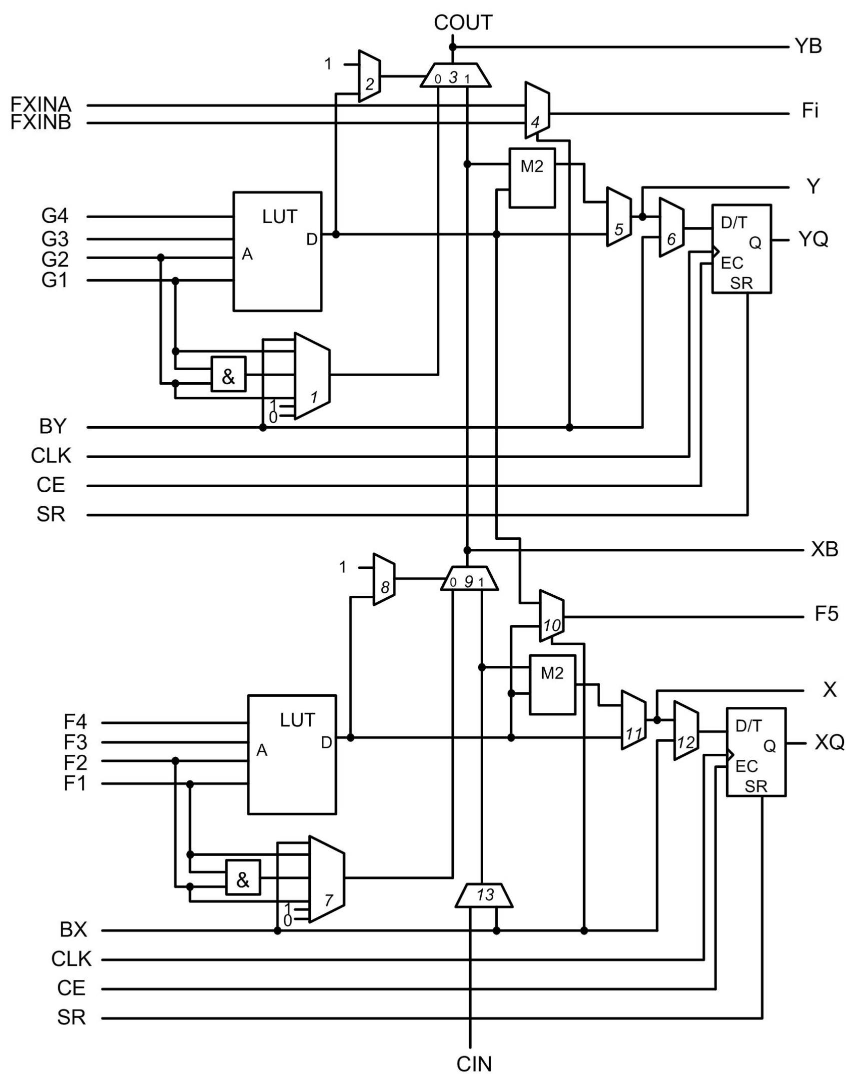

Рис. 11. Структура секции в блоке типа SLICEL

• сумматора с ускоренным переносом, реализующего сложение двухразрядных чисел (G2F2, G1F1) с учетом переноса в младший разряд (CIN). С помощью двух элементов блока LUT вырабатываются функции F1 XOR F2 и G1 XOR G2. Эти функции использу-

ются для формирования разрядов суммы на элементах сложения по модулю 2 с разрядом переноса, а также для формирования переноса в старший разряд с помощью мультиплексоров *3*, *9* и мультиплексоров выбора режима *1*, *2*, *7*, *8*;

• двух триггеров для хранения сигналов Y и X или сигналов BX и BY.

Сумматор с ускоренным переносом может также функционировать в качестве счетчика при выборе на мультиплексорах *1* и *7* одного из двух разрядов чисел G и F и подаче единицы на вход CIN. Другие входы мультиплексоров *1* и *7* служат для задания константных значений.

Блоки вводы вывода ПЛИС типа FPGA сложнее, чем в ПЛИС типа CPLD. Основное отличие заключается в наличии в этих блоках динамических триггеров для хранения входного и выходного сигналов, а также внутренних сигналов управления третьим состоянием сигналов. Кроме этого, в последних поколениях FPGA указанные регистры способны синхронизироваться от двух тактовых сигналов, что позволяет использовать технологию передачи с удвоенной скоростью (DDR). Для обеспечения возможности работы ПЛИС с несколькими типами интерфейсов, использующих при передаче различные уровни сигналов, блоки ввода/вывода подключают к различным цепям опорного напряжения. Группа блоков ввода/вывода, подключенных к одному и тому же сигналу опорного напряжения называется *банком ввода/вывода*. Существенное расширение области применения FPGA достигается также использованием программируемых схем параллельного и последовательного согласования волновых сопротивлений, соответствующих современным стандартам (PCI, PCI-X, LVCMOS, LVTTL, LVDS и др.).

Преимуществами современных ПЛИС являются:

• малое время и простота проектирования;

• низкая стоимость разработки;

• сокращение используемого пространства печатных плат;

• более низкая стоимость в расчете на одну микросхему по сравнению с заказными ИС;

• более продолжительное время обращения продукта на рынке благодаря возможности перепрограммирования;

• возможность создания динамически реконфигурируемых устройств.

К недостаткам можно отнести более низкую скорость работы ПЛИС по сравнению с полностью заказными ИС, а также нерентабельность использования в крупносерийном производстве.

Для более детального изучения архитектуры ПЛИС следует обратиться к документации, предоставляемой фирмами-производителями (например, материалы [1 – 3] доступны на сайте www.xilinx.com).

2. ОСНОВЫ ЯЗЫКА VHDL

2.1. Назначение языка VHDL

Язык VHDL (Very high speed integration circuits Hardware Description Language) является наиболее распространенным языком описания аппаратных средств. Он был принят в качестве стандарта в 1987 г. (стандарт VHDL-87), а с начала 1990-х годов описание на этом языке стали использовать при автоматизированном синтезе схем. Позднее был принят расширенный стандарт VHDL-93, а также стандарт VHDL-AMS для описания аналого-цифровых устройств (в данном пособии не рассматриваются). В настоящее время язык VHDL (наряду с языком Verilog) поддерживается многими САПР цифровых устройств. Язык VHDL используется в следующих целях:

• описание поведения цифровых устройств во времени и при изменении входных воздействий;

• описание структуры цифровых устройств с различной степенью детализации (на системном и блочном уровнях, на уровне регистровых передач, на уровне вентилей);

• моделирование цифровых устройств;

• описание тестовых воздействий при моделировании устройств;

• автоматизации преобразования исходного описания схемы в описание на более низком уровне (вплоть до вентильного).

По синтаксису язык VHDL в большой степени похож на универсальные языки программирования. Однако для описания сложных объектов, последовательно соединенные части которых могут функционировать параллельно, используются специфические элементы языка (сигналы, порты, конфигурации и т. д.). Особо под-

черкнем, что VHDL является проблемно-ориентированным языком. Поэтому следует осмотрительно применять практические навыки, полученные, например, при программировании на языках высокого уровня. Опытные разработчики аппаратных средств используют язык VHDL для быстрого и понятного описания схем, четко представляя при этом все проблемы их конкретного воплощения на основе ПЛИС или ASIC.

Описание на языке VHDL состоит из ключевых слов, объектов и операторов. Различают последовательные и параллельные операторы. Кроме этого, в настоящее время не все операторы языка могут быть преобразованы в реализуемое на аппаратном уровне описание (синтезированы), поэтому различают множества синтезируемых и несинтезируемых операторов. Как правило, при описании устройства можно использовать три различных стиля:

• поведенческий стиль, при котором для описания проекта используются причинно-следственные связи между событиями на входах устройства и событиями на его выходах (без уточнения структуры);

• структурный стиль, при котором устройство представляется в виде иерархии взаимосвязанных простых устройств (подобно стилю, принятому в схемотехнике);

• потоковый стиль описания устройства, основанный на использовании логических уравнений, каждое из которых преобразует один или несколько входных информационных потоков в выходные потоки.

В практике описания на языке VHDL часто используется комбинация указанных стилей: для каждого устройства или его части выбирается наиболее подходящий стиль описания. Перечисленные стили будут продемонстрированы и обсуждены на примерах.

Заметим, что изложение основ языка VHDL, приведенное в пособии, не может претендовать на полноту: синтаксические конструкции упрощены; незначительные, на взгляд автора, детали не представлены и не объяснены. Однако примеры, рассмотренные ниже, могут оказаться полезными при описании устройств с целью их последующего синтеза. Для таких целей используются шаблонные способы описания, заведомо обеспечивающие предсказуемые результаты. Для всестороннего изучения языка VHDL можно обратиться к стандарту VHDL-93 [5] или к русскоязычной литературе

[2 – 5]. Для глубокого понимания алгоритмов синтеза низкоуровневых описаний по исходному коду языка VHLD следует обратиться к документации, прилагаемой к программам-синтезаторам [6].

2.2. Объекты языка VHDL и их типы

Описание устройств с помощью языка VHDL основано на использовании объектов (не соответствуют понятию «объект» при объектно-ориентированном подходе к программированию). Объекты могут относиться к следующим категориям: константы; сигналы; переменные; файлы. Объект именуется с помощью правильного идентификатора и, как в языках программирования, обладает областью видимости. В языке VHDL применяется строгая типизация: для выполнения действий над разнотипными объектами требуется явно преобразовать их к одному и тому же типу. При этом предусмотрена возможность создавать на основе базовых типов новые пользовательские типы и их подтипы (с применением конструкций TYPE и SUBTYPE), создавать композиции типов (массивы и записи), а также определять для них процедуры преобразования.

Упрощенный синтаксис описания типа и подтипа:

```
<Тип>::= TYPE <Идентификатор> IS <Описание типа> ;
<Подтип>::= SUBTYPE <Идентификатор> IS <Описание подтипа> ;
```

Примеры описания типов и подтипов:

```
TYPE Multi_Level_Logic IS (LOW, HIGH, RISING, FALLING,
AMBIGUOUS);
TYPE BIT IS ('0','1') ;
TYPE BYTE_LENGTH_INTEGER IS RANGE 0 TO 255;
TYPE WORD_INDEX IS RANGE 31 DOWNTO 0;
SUBTYPE HIGH_BIT_LOW IS BYTE_LENGTH_INTEGER RANGE 0
TO 127;
TYPE MY_WORD IS ARRAY (0 TO 31) OF BIT ;
TYPE STATE_TYPE IS (s0,s1,s2,s3);
```

Базовые типы данных

Базовые типы данных, описанные в библиотеке STANDARD (автоматически подключается для всех проектов во всех САПР), перечислены в табл. 1.

Таблица 1

Базовые типы данных VHDL-93

Тип данных	Значения/Диапазон
Перечислимые типы	
BOOLEAN	TRUE, FALSE
BIT	‘0’,’1’
BIT_VECTOR	Массив элементов BIT с индексами от 0 до +2147483647
SEVERITY_LEVEL	NOTE, WARNING, ERROR, FAILURE
FILE_OPEN_KIND	READ_MODE, WRITE_MODE, APPEND_MODE
FILE_OPEN_STATUS	OPEN_OK, STATUS_ERROR, NAME_ERROR, MODE_ERROR
Целочисленные типы	
INTEGER	–2147483647 … +2147483647
POSITIV	1… +2147483647
NATURAL	0… +2147483647
Типы чисел с плавающей запятой	
REAL	–1.0E38 … +1.0E38
Символьные типы	
CHARACTER	Nul, …,’0’, …, ’9’, ’@’, ’:’, ’;’, …, ’A’, …, ’Z’, ’a’, …, ’z’, DEL
STRING	Массив элементов CHARACTER с индексами от 1 до +2147483647
Физические типы	
TIME	–2147483647 … +2147483647
DELAY_LENGTH	0…+2147483647

Атрибуты

Помимо своего значения объекты в языке VHDL обладают также атрибутами, отражающими свойства этих объектов. Для каждого объекта заранее предопределенны несколько атрибутов, кроме того, можно создавать пользовательские атрибуты (синтезаторами

не поддерживаются). Для создания пользовательского атрибута необходимо его объявить и специфицировать.

Объявление атрибута:

<Объявление Атрибута> ::=
ATTRIBUTE <Идентификатор>: <Тип>;

Спецификация атрибута:

<Спецификация атрибута> ::=
ATTRIBUTE <Идентификатор> **OF** <Описание> : <Класс>
IS <Выражение>;
<Описание> ::= <имя> [{ , <Имя> }] | **OTHERS** | **ALL**
<Класс> ::= **ENTITY** | **ARCHITECTURE** | **CONFIGURATION**
| **PROCEDURE** | **FUNCTION** | **PACKAGE**
| **TYPE** | **SUBTYPE** | **CONSTANT**
| **SIGNAL** | **VARIABLE** | **COMPONENT**
| **LABEL** | **LITERAL** | **UNITS**
| **GROUP** | **FILE**
<Имя> ::= <таг> [<Сигнатура>]
<Таг> ::= <Идентификатор> | <Символьный Литера> |
<Символ Оператора>

Опции OTHERS и ALL позволяют задать список описаний не в виде перечисления существующих, а с помощью выбора с исключением (OTHERS) или выбора всех (ALL).

Пример объявления и спецификации атрибутов:

```
ATTRIBUTE state_vector : string;
ATTRIBUTE state_vector OF fsm : ARCHITECTURE IS
"current_state";
```

Константы

Константы — это объекты, значение которых не изменяется в процессе работы устройства. При объявлении может быть указано выражение, определяющее значение константы:

<Константа> ::=
CONSTANT <список идентификаторов> : <тип> [:= <выражение>]

Пример декларации константы типа TIME:

```
CONSTANT A: TIME := 30ns;
```

Сигналы

Сигналы — это объекты, обладающие историей изменения (прошлым и текущим состояниями). В процессе моделирования возникают события, которые являются причинами возникновения новых событий. При возникновении события, приводящего к изменению сигнала, это изменение происходит не мгновенно, а через некоторое оговоренное время (большее или равное дельта-задержке). Это свойство сигнала позволяет учесть при моделировании причинно-следственные связи между воздействием и откликом.

Объявление сигнала:

<Сигнал> ::=
SIGNAL <Список идентификаторов> : <Тип> [<Вид>]
[:= <Выражение>] ;
<Вид сигнала> ::= **REGISTER** | **BUS**

Для каждого сигнала предопределены следующие атрибуты, поддерживаемые программами синтеза (на примере сигнала s):

• s'**stable**[(T)] — атрибут равен значению true, если за время T сигнал не изменял своего состояния;

• s'**transaction** — атрибут типа bit, изменяющий состояние на противоположное при каждом присваивании нового значения сигналу s;

• s'**event** — атрибут, равный true, если в данном цикле моделирования произошло изменение сигнала s;

• s'**active** — атрибут, равный значению true, если в данном цикле моделирования произошло присваивание нового значения сигналу s;

• s'**last_value** — атрибут, равный сигналу s, до момента его последнего изменения.

Пример декларации и инициализации сигнала:

```
SIGNAL B: BIT_VECTOR(3 DOWNTO 0):="0000";
```

Пример присвоения значения сигналу:

```
B<="0001";
```

Пример использования атрибутов сигнала:

```
IF CLK'event AND CLK='1' THEN
B<=D; --Изменить значение по фронту синхросигнала
END IF;
```

Переменные

Значения переменных также могут изменяться в процессе работы устройства. Однако при моделировании изменения значений переменных происходят «мгновенно» и в том порядке, в котором присвоения встречаются в коде.

Объявление переменной:

```
<Переменная> ::=
[ SHARED ] VARIABLE <Список идентификаторов> : <Тип> [ :=
<Выражение> ] ;
```

Опция SHARED используется при объявлении переменных в интерфейсах и архитектурах устройств, интерфейсах и телах пакетов, а также в блоках.

Пример декларации переменной:

```
VARIABLE C: INTEGER := 100;
```

Пример присвоения значения переменной:

```
B:=200;
```

Файлы

Файл — это объект, позволяющий в процессе моделирования производить операции чтения и записи физического файла. Данный объект относится к несинтезируемому подмножеству языка VHDL.

Сигналы типа STD_LOGIC и STD_LOGIC_VECTOR

Наиболее широко на практике применяются типы STD_LOGIC и STD_LOGIC_VECTOR, описанные в библиотеке STD_LOGIC_1164. Для данных типов в библиотеке реализованы многочисленные функции преобразования, арифметические и логические функции, а также функции разрешения. *Функцией разрешения* на-

зывается функция, по которой определяется результирующее значение сигнала, управляемого несколькими источниками. При схемной реализации программа синтеза по указанию разработчика установит для таких сигналов дополнительный элемент, позволяющий реализовать эквивалентную функцию (например, дизъюнктор). Сигнал типа STD_LOGIC принимает одно из девяти значений:

- 'U' — не инициализировано;
- 'X' — неизвестное (сильное значение);
- '0' — логический нуль (сильное значение);
- '1' — логическая единица (сильное значение);
- 'Z' — третье состояние;
- 'W' — неизвестное (слабое значение);
- 'L' — логический нуль (слабое значение);
- 'H' — логическая единица (слабое значение);
- '-' — неопределенное значение.

Слабые сигналы имеют меньший приоритет по сравнению с сильными. Значения 'U', 'X', 'W', '-' служат для моделирования некорректных состояний устройств.

Библиотека STD_LOGIC_1164 поддерживается современными программами синтеза.

Примеры использования сигналов STD_LOGIC и STD_LOGIC_VECTOR:

```
CLK :IN std_logic;
RST :IN std_logic;
CH_CODE :INOUT std_logic;
DATA_BUS :INOUT std_logic_vector(w-1 DOWNTO 0);
```

Операции языка VHDL

Над объектами стандартных типов, как над переменными в языках программирования, можно выполнять арифметические и логические операции, объединяя их в выражения. Операции языка VHDL для типов BIT и BOOLEAN перечислены в табл. 2 в порядке возрастания их приоритета.

Примеры выполнения операций над объектами:

```
result <= a + b;--сумматор (поведенческий вариант)
sum <= (a XOR b) XOR cin;--сумматор (структурный вариант)
cout <= (a AND b) OR (a AND cin) OR (b AND cin);
equal <= '1' WHEN a = b ELSE '0'; --компаратор
outp <= "00000001" SLL conv_integer(inp); --дешифратор
```

Таблица 2

Операции языка VHDL

Тип операций	Обозначение	Комментарий
Логические	AND OR NAND NOR XOR XNOR	Логическое И Логическое ИЛИ Логическое И-НЕ Логическое ИЛИ-НЕ Исключающее ИЛИ Эквивалентность
Сравнения	= /= < > <= >=	Равенство Неравенство Меньше Больше Меньше или равно Больше или равно
Сдвига	SLL SRL SLA SRA ROL ROR	Сдвиг влево логический Сдвиг вправо логический Сдвиг влево арифметический Сдвиг вправо арифметический Сдвиг влево циклический Сдвиг вправо циклический
Знака и сложения	+ – &	Сложение (знак плюс) Вычитание (знак минус) Конкатенация
Умножения	* / MOD REM	Умножение Деление[1] Модуль[1] Остаток[1]
Смешанные	ABS NOT **	Абсолютное значение Отрицание Степень

[1] Операции синтезируются ограниченно.

2.3. Интерфейс и архитектура устройств

Устройство или его часть описывается на языке VHDL с помощью двух связанных конструкций: интерфейса устройства (ENTITY) и архитектуры устройства (ARCHITECTURE). Конструкции ENTITY могут соответствовать несколько вариантов конструкции ARCHITECTURE, описывающих поведение или структуру устройства. Если описаний ARCHITECTURE несколько, то необходимо для каждого используемого экземпляра устройства указать требуемое описание. Устройства могут быть организованы по иерархическому принципу, т. е. могут состоять из других устройств. Связность частей устройства задается с помощью сигналов.

Упрощенный синтаксис конструкции ENTITY:

<Интерфейс> ::=
ENTITY <Идентификатор**> IS**
[**GENERICS** <Список настроечных констант>]
[**PORT** <Список портов>]
    <Декларативная часть>
[BEGIN
    <Пассивные конструкции языка> **]**
END [ENTITY] [<Идентификатор>**] ;**

В конструкции ENTITY предусмотрена возможность задавать список настроечных параметров с помощью ключевого слова GENERIC. Это позволяет описать универсальное устройство, некоторые числовые параметры которого (например, ширина шины данных и адреса) будут определены при его включении в проект. Декларация портов PORT позволяет описать входные объекты (IN) только для чтения, выходные объекты (OUT) — только для изменения, двунаправленные объекты (INOUT) — для чтения и изменения, двунаправленные объекты (BUFFER) — для чтения и изменения только одним источником. Объект в режиме LINKAGE может читаться или изменяться только подстановкой в качестве фактического параметра для другого объекта в режиме LINKAGE.

Упрощенный синтаксис декларации PORT:

<Описание портов>::=
PORT(<Описание порта> {;<Описание порта>});
<Описание порта>::= <Список идентификаторов> :
<Режим> <Тип>
<Режим> ::= **BUFFER**|**IN**|**OUT**|**INOUT**|**LINKAGE**

Пример конструкции ENTITY:

```
ENTITY Decoder IS
        PORT ( inp: IN std_logic_vector(2 DOWNTO 0);
        outp: OUT std_logic_vector(7 DOWNTO 0));
END Decoder;
```

В декларативной части ENTITY описываются объекты, типы, подпрограммы и прочие элементы, которые будут использованы при описании устройства. Пассивные конструкции языка не синтезируются. Они позволяют выполнять различные проверки в процессе моделирования.

Структура или поведение устройств описываются с помощью конструкции ARCHITECTURE.

Упрощенный синтаксис ARCHITECTURE:

```
<Архитектура> ::=
ARCHITECTURE <Идентификатор> OF <Интерфейс> IS
        <Декларативная часть>
BEGIN
        <Параллельные операторы>
END [ ARCHITECTURE ] [<Идентификатор>] ;
```

В декларативной части ARCHITECTURE описываются объекты, типы, подпрограммы и прочие элементы, которые будут использованы при описании устройства. После ключевого слова BEGIN с помощью параллельных операторов описываются параллельно работающие части устройства.

Пример описания устройства:

```
-- Комментарий: Интерфейс полного сумматора
ENTITY Full_Adder IS
PORT (          X, Y, Cin:IN Bit;
                Cout, Sum:OUT Bit);
END Full_Adder ;
-- Комментарий: Архитектура 1 полного сумматора (потоковый стиль)
ARCHITECTURE DataFlow1 OF Full_Adder IS
SIGNAL A,B: Bit;
BEGIN
        A <= X XOR Y;
        B <= A AND Cin;
        Sum <= A XOR Cin;
```

```
        Cout <= B OR (X AND Y);
END ARCHITECTURE DataFlow1 ;
-- Комментарий: Архитектура 2 полного сумматора (потоковый стиль)
ARCHITECTURE DataFlow2 OF Full_Adder IS
BEGIN
        Sum <= (X XOR Y) XOR Cin;
        Cout <= (X AND Y) or (X AND Cin) OR (Y AND Cin);
END DataFlow2;
```

2.4. Пакеты и библиотеки

Для описания простого устройства достаточно в единственном файле проекта описать хотя бы один интерфейс и его архитектуру. При проектировании же сложных устройств, которые предполагается использовать многократно, целесообразно компоновать описания в пакеты (PAKAGE). Декларация пакета состоит из описания интерфейса и описания тела:

```
<Интерфейс Пакета> ::=
PACKAGE <имя> IS
        <Декларативная часть>
END [ PACKAGE ] [ <имя> ] ;
<Тело Пакета> ::=
PACKAGE BODY <имя> IS
        <Декларативная часть>
END [ PACKAGE ] [ <имя> ] ;
```

В декларативной части интерфейса могут быть объявлены подпрограммы, типы, подтипы, объекты всех видов, атрибуты, компоненты и т. д. В теле пакета приводятся реализации процедур и указываются значения констант, декларированных в интерфейсной части:

```
PACKAGE TriState IS
        TYPE Tri IS ('0', '1', 'Z', 'E');
        FUNCTION BitVal (Value: Tri) RETURN Bit;
        FUNCTION TriVal (Value: Bit) RETURN Tri;
        TYPE TriVector IS ARRAY (Natural RANGE <>) OF Tri;
        FUNCTION Resolve (Sources: TriVector) RETURN Tri;
END PACKAGE TriState;
PACKAGE BODY TriState IS
```

```
        FUNCTION BitVal (Value: Tri) RETURN Bit IS
                CONSTANT Bits : Bit_Vector := "0100";
        BEGIN
                RETURN Bits(Tri'Pos(Value));
        END;
        FUNCTION TriVal (Value: Bit) RETURN Tri IS
        BEGIN
                RETURN Tri'Val(Bit'Pos(Value));
        END;
        FUNCTION Resolve (Sources: TriVector) RETURN Tri IS
                VARIABLE V: Tri := 'Z';
        BEGIN
                FOR i IN Sources'Range LOOP
                        IF Sources(i) /= 'Z' THEN
                                IF V = 'Z' THEN
                                        V := Sources(i);
                                ELSE
                                        RETURN 'E';
                                END IF;
                        END IF;
                END LOOP;
                RETURN V;
        END;
END PACKAGE BODY TriState ;
```

Пакеты, в свою очередь, объединяются в библиотеки (LIBRARY). Для подключения библиотеки необходимо использовать оператор подключения библиотеки и использования пакета:

> <Подключение библиотеки> ::=
> **LIBRARY** <имя библиотеки> { , <Имя библиотеки> };
> <Использование пакета>::=
> <Имя библиотеки>.<Имя пакета>.<Символьное имя> |
> <Идентификатор> | **ALL**;

Пример использования библиотеки и пакета:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
```

2.5. Параллельные операторы

Операторы данной группы описывают взаимосвязанные параллельно функционирующие части устройства. Параллельные операторы выполняются асинхронно по отношению друг к другу. К таким операторам относятся:

- блоки (BLOCK);
- процессы (PROCESS);
- параллельные присваивания сигналов (<=);
- параллельные вызовы процедур;
- параллельные операторы-ловушки (ASSERT);
- экземпляры компонентов (COMPONENT).

Оператор BLOCK

Этот оператор позволяет описать группу параллельных операторов:

<Блок> ::=
[<метка>] : **BLOCK** [(<охранное выражение>)] **[IS]**
    [**GENERIC** (<объявление настроечных констант>);]
    [**GENERIC MAP** (список Связывания настроечных констант);]
    [**PORT** (<объявление Портов>);]
    [**PORT MAP** (<список Связывания Портов>)]
    <Декларативная часть>
BEGIN
    <параллельные операторы>
END BLOCK [<метка>];

Охранное выражение позволяет задать скрытый локальный сигнал GUARD, который можно использовать в пределах блока в качестве разрешающего сигнала при присваиваниях. Объявление настроечных констант и их связывание конкретизируют настроечные значения. Объявление портов и их связывание позволяют ассоциировать порты схемы с внешними сигналами.

Пример описания устройства с использованием иерархии блоков:

```
C: BLOCK
BEGIN
        X: BLOCK
```

```
        PORT (P1, P2: INOUT BIT); --объявление портов
        PORT MAP (P1 => S1, P2 => S2); --связывание портов
        CONSTANT Delay: DELAY_LENGTH := 1 ms; --декларации
        SIGNAL P3: BIT;
        BEGIN
                P3 <= P1 AFTER Delay; --присвоение
                •
                B: BLOCK
                •
                BEGIN
                        •
                END BLOCK B;
        END BLOCK X;
END BLOCK C;
```

Оператор PROCESS

Этот оператор описывает независимые группы последовательных операторов в виде параллельных процессов.

Упрощенный синтаксис оператора PROCESS:

```
<Процесс>::=
[<метка:>][POSTPONED] PROCESS
(<список чувствительности>) [IS]
    <Декларативная часть>
BEGIN
    <Последовательные операторы>
END [POSTPONED] PROCESS [<метка>] ;
```

Использование ключевого слова POSTPONED позволяет при моделировании указать на отложенный процесс (исполнение его отложено относительно всех исполняющихся в данный момент времени *t* процессов). Такой процесс исполняется последним для момента времени *t*. Декларативная часть содержит объявления объектов, типов, атрибутов, а также тела подпрограмм и альтернативные точки входа в подпрограммы.

Список чувствительности представляет собой перечень сигналов, позволяющий моделирующей программе определить моменты времени, в которые процесс должен активизироваться. При синтезе схемы список чувствительности не учитывается.

Пример описания динамического триггера с использованием процесса:

```
FF: PROCESS (CLK,RST,WE)
BEGIN
   IF RST='1' THEN
        D<='0';
   ELSIF ((CLK='1') AND (CLK'event)) THEN
          IF (WE='1') THEN
            D<=D_IN;
          END IF;
   END IF;
END PROCESS;
```

Оператор параллельного присваивания сигнала

Этот оператор эквивалентен процессу, в котором при определенных условиях происходит присваивание сигналу нового значения.

Синтаксис оператора параллельного присваивания сигнала:

<Параллельное присваивание сигнала>::=
[<метка> :] [**POSTPONED**] <условное присваивание сигнала>
| [<метка> :] [**POSTPONED**] <присваивание сигнала>
<условное присваивание сигнала> ::=
<сигнал><= <Опции> { <образец> **WHEN** <условие> **ELSE** }
<образец> [**WHEN** <условие>];
<присваивание сигнала> ::=
WITH <выражение> **SELECT**
<сигнал> <= <Опции> { <образец> **WHEN** <значение>, }
<образец> **WHEN** <значение>;
<Опции> ::= [**GUARDED**] [<способ задержки>]
<образец> ::=
<выражение> [**AFTER** <время>]| **NULL** [**AFTER** <время>]

Опция GUARDED обеспечивает использование при присваивании дополнительного разрешающего сигнала с именем GUARD (неявно объявленного в блоке или явно объявленного пользователем). Все присваивания разрешены по фронту или по уровню сигнала GUARD. Опция AFTER определяет задержку изменения сигнала.

Динамический триггер можно описать с помощью оператора параллельного присваивания сигнала следующим способом:

```
ENTITY ff IS
        PORT (RST,CLK,WE: IN std_logic;
          D_IN: IN std_logic;
          D: OUT std_logic);
END ff;
ARCHITECTURE DataFlow OF ff IS
BEGIN
D<=     '0'       WHEN RST='1' ELSE
        D_IN      WHEN (CLK='1') AND (CLK'event) AND (WE='1');
END DataFlow;
```

Описание дешифратора:

```
LIBRARY ieee; --Описание подключаемых библиотек
USE ieee.std_logic_1164.ALL;
ENTITY decoder IS
        PORT ( inp: IN std_logic_vector(2 DOWNTO 0);
          outp: OUT std_logic_vector(7 DOWNTO 0));
END decoder;
ARCHITECTURE DataFlow OF decoder IS
BEGIN
        outp(0) <= '1' WHEN inp = "000" ELSE '0';
        outp(1) <= '1' WHEN inp = "001" ELSE '0';
        outp(2) <= '1' WHEN inp = "010" ELSE '0';
        outp(3) <= '1' WHEN inp = "011" ELSE '0';
        outp(4) <= '1' WHEN inp = "100" ELSE '0';
        outp(5) <= '1' WHEN inp = "101" ELSE '0';
        outp(6) <= '1' WHEN inp = "110" ELSE '0';
        outp(7) <= '1' WHEN inp = "111" ELSE '0';
END DataFlow;
```

Процедуры и функции

Другим способом группировки последовательных действий являются процедуры и функции, которые могут быть вызваны и исполнены в виде параллельных процессов. Объявленные внутри процедуры или функции объекты, как и для других способов груп-

пировки (PROCESS и BLOCK), не могут быть использованы за их пределами, т. е. приобретают свойство локальности.

Упрощенный синтаксис объявления процедуры:

<Процедура> ::=
PROCEDURE <Имя> **[** (<Список формальных параметров >) **]**
|**[PURE**|**IMPURE]** **FUNCTION <**Имя> **[**(< Список формальных параметров>) **]**
RETURN <Тип>

Пример объявления и параллельного вызова процедуры:

```
ENTITY sort4 IS
   GENERIC (top : integer :=3);
   PORT (
         a, b, c, d : IN bit_vector(0 TO top);
         ra, rb, rc, rd : OUT bit_vector(0 TO top)
    );
END sort4;

ARCHITECTURE muxes OF sort4 IS
   PROCEDURE sort2(SIGNAL x, y : IN bit_vector(0 TO top);
         SIGNAL g, l : OUT bit_vector(0 TO top)
   ) IS
   BEGIN
     IF x > y THEN
         g <= x;
         l <= y;
     ELSE
         l <= x;
         g <= y;
END IF;
     END sort2;
SIGNAL v1,v2,v3,v4,v5,v6,v7,v8,v9,v10 : bit_vector(0 TO top);

BEGIN
--Параллельные вызовы процедуры
         sort2(a, c, v1, v2);
         sort2(b, d, v3, v4);
         sort2(v1, v3, v5, v6);
         sort2(v2, v4, v7, v8);
         sort2(v6, v7, v9, v10);
```

```
-- Параллельные присваивания сигналов
        ra <= v8;
        rb <= v10;
        rc <= v9;
        rd <= v5;
END muxes;
```

Оператор-ловушка ASSERT

Для упрощения отладки в языке VHDL используется оператор-ловушка ASSERT, позволяющий выполнить в ходе моделирования проверочное сравнение и при обнаружении ошибки (ложном значении выражения) выдать заданное сообщение:

> <Сообщение> ::=
> [<метка> :] [**POSTPONED**] **ASSERT** <выражение>
> [**REPORT** <сообщение>] [**SEVERITY** <тип сообщения>];

Пример сообщения:

```
ASSERT D ='1'
REPORT "Ошибка, D не равно 1";
```

Использование компонентов

Для использования устройства в виде компонента другого устройства необходимо объявить его в декларативной части конструкции ARCHITECTURE:

> <Объявление компонента> ::=
> **COMPONENT** <Имя компонента>
> **[GENERIC** <Список настроечных параметров>;]
> **[PORT** <Список портов>;]
> **END COMPONENT**;

После этого можно создать экземпляр компонента в описательной части ARCHITECTURE с помощью конструкции использования:

> <Экземпляр компонента> **::=**
> <Метка экземпляра>: <Имя компонента>

[GENERIC MAP (<Значения настроечных констант>)**]**
[PORT MAP (<Список соответствий сигналов>)**]**;

Пример использования настроечных констант и декларации компонентов:

```
-- Пример использования компонентов
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY adder IS
    GENERIC ( n:                INTEGER:=16);
    PORT (    a,b: IN std_logic_vector (n-1 DOWNTO 0);
        result: OUT std_logic_vector(n-1 DOWNTO 0));
END adder;
ARCHITECTURE behave OF adder IS
BEGIN
        result <= a + b;
END;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY adders IS
    GENERIC(          op_n:     INTEGER := 16;
        sum_n:   INTEGER :=16);
    PORT(     a,b,c: IN std_logic_vector(op_n-1 DOWNTO 0);
        result: OUT std_logic_vector(sum_n-1 DOWNTO 0));
END adders;
ARCHITECTURE behave OF adders IS
    COMPONENT adder
                GENERIC ( n:INTEGER:=op_n);
        PORT ( a,b: IN std_logic_vector(n-1 DOWNTO 0);
            result: OUT std_logic_vector(n-1 DOWNTO 0));
    END COMPONENT;
    SIGNAL rez: std_logic_vector(sum_n-1 DOWNTO 0);

BEGIN
        a1:adder
            PORT MAP (          a => a, b=>b, result=>rez);
```

```
        a2:adder
            PORT MAP ( a=>rez,
                               b=>c,
                        result => result);
END behave;
```

Оператор GENERATE

Этот оператор обеспечивает возможность итеративного или условного повторения деклараций и операторов:

```
<Генерация> ::=
[<Метка>]:<Способ генерации> GENERATE
    [<Декларации> BEGIN]
    [<Параллельные операторы>]
END GENERATE [<Метка>];
<Способ генерации> ::= FOR <Идентификатор> IN
<Диапазон> | IF <Условие>
```

При итеративной генерации используется конструкция FOR, после которой указываются идентификатор номера итерации и его диапазон изменения. При использовании условной генерации благодаря конструкции IF генерация выполняется, если условие истинно.

Пример конструкции GENERATE:

```
Gen: BLOCK --двоичное дерево
BEGIN
        L1: CELL PORT MAP (Top, TBus(1), TBus(2)) ; --Top,Left,Right
        L2: FOR I IN 1 TO 7 GENERATE
                L3: FOR J IN 0 TO 2**I-1 GENERATE
                L5: CELL PORT MAP (TBus(2**I+J-1),
TBus((2**I+J)*2-1), TBus(2**I+J)*2);
                END GENERATE ;
        END GENERATE ;
END BLOCK Gen;
```

2.6. Последовательные операторы

Такие операторы используются для описания алгоритма функционирования параллельных процессов и подпрограмм. Их исполнение происходит в той последовательности, в которой они описаны в программе. Последовательными операторами являются:

• оператор WAIT;
• оператор-ловушка ASSERT и оператор сообщения REPORT;
• оператор присваивания сигнала;
• оператор присваивания переменной;
• вызов процедуры;
• оператор IF;
• оператор CASE;
• оператор цикла LOOP;
• оператор NEXT;
• оператор EXIT;
• оператор RETURN;
• пустой оператор NULL.

Оператор ожидания WAIT

Этот оператор дает возможность приостановить исполнение процесса или подпрограммы:

<Ожидание> ::=
[<Метка :] **WAIT** [**ON** <Список чувствительности>]
[**UNTIL** <Условие>] [**FOR** <Задержка>];

Опция ON позволяет описать список сигналов, изменение состояния которых ожидается; опция UNTIL — описать ожидание сложного события, выраженного в виде условия; опция FOR — описать временнýю задержку.

Пример использования оператора WAIT:

```
WAIT ON Clk;
```

Последовательный оператор-ловушка ASSERT

Этот оператор, как и параллельный оператор-ловушка, используется в качестве средства отладки в процессе моделирования:

<Ловушка> ::=
[<Метка> :] **ASSERT** <Выражение> [**REPORT** <Сообщение>]
[**SEVERITY** <Тип соосбщения>];

Сообщение будет выдано при ложном значении выражения. Тип сообщения влияет на дальнейший ход моделирования. Конст-

рукция <Тип сообщения> может принимать значения: NOTE, WARNING, ERROR, FAILURE.

Оператор сообщения может использоваться независимо от оператора ASSERT:

```
<Сообщение> ::=
[ Метка :] REPORT <Строка сообщения>
[ SEVERITY <Тип сообщения> ] ;
```

Пример использования операторов ASSERT и REPORT:

```
ASSERT (IRDY_N = '0')
REPORT "Target device: FRAME signal deassertion error.
IRDY is not asserted."
SEVERITY ERROR;
```

Последовательный оператор присваивания сигнала

Этот оператор служит для изменения значения сигнала и используется внутри процесса.

Синтаксис оператора присваивания сигнала:

```
<Присваивание сигнала> ::=
[ <Метка> :] <Сигнал> <= [ <Способ задержки> ] <Образец> ;
<Способ задержки> ::= TRANSPORT
|[REJECT <время> ] INERTIAL
<Образец> ::=
<Выражение> [ AFTER <время> ]| NULL [ AFTER <Время> ]
```

Необязательная опция TRANSPORT позволяет описать транспортную задержку изменяемого сигнала относительно сигнала-образца подобно тому, как сигналы задерживаются в линиях передачи. Частота срабатывания описанных таким образом устройств бесконечна, а любой импульс передается вне зависимости от его длительности. Если не указан способ задержки или указано ключевое слово INERTIAL, задержка считается инерционной. Это означает, что устройство, реализующее указанную функцию присваивания, не будет передавать импульс, длительность которого меньше времени переключения схемы устройства. Опция REJECT позволяет описать минимальную длительность импульса сигнала-образца, приводящего к изме-

нению сигнала. При импульсах меньшей длительности присваивание не производится.

Пример присваивания сигнала:

```
P3 <= P1 AFTER DELAY;
OUTPUT_PIN <= TRANSPORT INPUT_PIN AFTER 10 NS;
```

Оператор присваивания переменной

Оператор присваивания переменной в языке VHDL подобен оператору присваивания в языках программирования.

Синтаксис оператора присваивания переменной:

```
<ПРИСВАИВАНИЕ ПЕРЕМЕННОЙ> ::=
[ <МЕТКА> :] <ПЕРЕМЕННАЯ> := <ВЫРАЖЕНИЕ>;
```

Пример присваивания переменной:

```
i := i - 1;
```

Оператор вызова процедуры

Этот оператор служит для исполнения алгоритма, описанного в процедуре или функции:

```
<ВЫЗОВ ПОДПРОГРАММЫ> ::=
[ <МЕТКА> : ] <ИМЯ ПРОЦЕДУРЫ ИЛИ ФУНКЦИИ>
[ ( <СПИСОК ФОРМАЛЬНЫХ ПАРАМЕТРОВ>) ];
```

Условный оператор IF

Условный оператор IF позволяет описать устройство, в алгоритме работы которого по результатам проверки заданных условий разрешается исполнение одной (или нуля) групп последовательных операторов. Выполнение последовательных операторов разрешается, если заданное условие истинно.

Синтаксис условного оператора:

```
<УСЛОВНЫЙ ОПЕРАТОР> ::=
[ <МЕТКА> :] IF <УСЛОВИЕ> THEN
    <ПОСЛЕДОВАТЕЛЬНЫЕ ОПЕРАТОРЫ>
```

```
{ ELSIF <Условие> THEN
      <Последовательные операторы>}
[ ELSE
      <Последовательные операторы>]
END IF [ <Метка> ];
```

Пример описания шифратора с помощью условного оператора:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY encoder IS
        PORT ( in1 :IN std_logic_vector(7 DOWNTO 0);
        out1 :OUT std_logic_vector(2 DOWNTO 0));
END encoder;
ARCHITECTURE behave OF encoder IS
BEGIN
    PROCESS (in1)
    BEGIN
        IF       in1(7) = '1' THEN out1 <= "111";
        ELSIF in1(6) = '1' THEN out1 <= "110";
        ELSIF in1(5) = '1' THEN out1 <= "101";
        ELSIF in1(4) = '1' THEN out1 <= "100";
        ELSIF in1(3) = '1' THEN out1 <= "011";
        ELSIF in1(2) = '1' THEN out1 <= "010";
        ELSIF in1(1) = '1' THEN out1 <= "001";
        ELSIF in1(0) = '1' THEN out1 <= "000";
        ELSE out1 <= "XXX";
           END IF;
    END PROCESS;
END behave;
```

Оператор выбора CASE

Этот оператор разрешает исполнение одного из многих альтернативных блоков последовательных операторов. Выбор блока среди альтернатив выполняется при совпадении значений выражения и образца.

Синтаксис оператора выбора:

```
<Выбор> ::=
[ <Метка> : ]
CASE <Выражение> IS
```

WHEN <Образец> => <Последовательные операторы>
{ **WHEN** <Образец> => <Последовательные операторы>}
END CASE [<Метка>] ;

Пример использования оператора выбора при описании дешифратора:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY decoder IS
        PORT ( inp: IN std_logic_vector(2 DOWNTO 0);
  outp: OUT std_logic_vector(7 DOWNTO 0));
END decoder;
ARCHITECTURE behave OF decoder IS
  BEGIN
  PROCESS (inp) BEGIN
        CASE inp IS
                WHEN "000" => outp <= "00000001";
                WHEN "001" => outp <= "00000010";
                WHEN "010" => outp <= "00000100";
                WHEN "011" => outp <= "00001000";
                WHEN "100" => outp <= "00010000";
                WHEN "101" => outp <= "00100000";
                WHEN "110" => outp <= "01000000";
                WHEN "111" => outp <= "10000000";
                WHEN OTHERS => outp <= "XXXXXXXX";
        END CASE;
  END PROCESS;
END behave;
```

Оператор цикла LOOP

Этот оператор позволяет описать часть устройства, алгоритм работы которой представляет собой повторяющиеся действия:

<Цикл> ::=
[<Метка> :]
[**WHILE** <Условие> | **FOR** <Идентификатор> **IN** <диапазон>]
LOOP
    <Последовательные операторы>
END LOOP [<Метка>];

Пример использования оператора цикла при описании шифратора:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.conv_std_logic_vector;
ENTITY encoder IS
  PORT ( a, b, c, d, e, f, g, h : IN std_logic;
  out2, out1, out0 : OUT std_logic);
END encoder;
ARCHITECTURE behave OF encoder IS
BEGIN
  PROCESS (a, b, c, d, e, f, g, h)
  VARIABLE inputs : std_logic_vector (7 DOWNTO 0);
  VARIABLE i : INTEGER ;
  BEGIN
    INPUTS := (h, g, f, e, d, c, b, a);
    i := 7;
    WHILE i >= 0 AND inputs(i) /= '1' LOOP
      i := i - 1;
    END LOOP;
    IF ( i < 0) THEN
      i := 0;
    END IF;
        -- conv_std_logic_vector (i, 3) - функция преобразования
        -- переменной типа integer в сигнал типа std_logic_vector
        -- Второй аргумент определяет размер вектора.
    (out2, out1, out0) <= conv_std_logic_vector (i, 3);
  END process;
END behave;
```

Операторы NEXT, EXIT, RETURN, NULL

Оператор NEXT применяется для того, чтобы завершить выполнение текущей итерации цикла LOOP:

<Следующий> ::=
[<Метка>:] **NEXT** [<Метка цикла>] [**WHEN** <Условие>] ;

Текущая итерация цикла завершается, если условие истинно или если конструкция WHEN не используется.

Оператор EXIT применяется для того, чтобы остановить выполнение цикла LOOP, в котором этот оператор находится:

<Выход>::=
[<Метка> :] **EXIT** [<Метка цикла>] [**WHEN** <Условие>] ;

Оператор возврата RETURN используется для завершения выполнения функции или процедуры и возврата результата.

Возврат ::=
[Метка :] **RETURN** [Выражение] ;

Пример использования оператора EXIT и RETURN при описании настраиваемого конъюнктора:

```
ENTITY AndGate IS
        GENERIC (N: Natural := 2);
        PORT ( Inputs: IN Bit_Vector (1 TO N);
                 Result: OUT Bit) ;
END entity AndGate ;
ARCHITECTURE Behavior OF AndGate IS
BEGIN
        PROCESS (Inputs)
        VARIABLE Temp: Bit;
        BEGIN
                 Temp := '1';
                 FOR i IN Inputs'range LOOP
                          IF Inputs(i) = '0' THEN
                                   Temp := '0';
                                   EXIT;
                          END IF;
                 END LOOP;
                 Result <= Temp AFTER 10 ns;
        END PROCESS;
END BEHAVIOR;
```

Пустой оператор NULL используется для явного указания на то, что никакие действия не выполняются. Наиболее оправданно использование NULL внутри оператора выбора CASE.

2.7. Пример описания устройств с тремя состояниями выходов

Потребность в устройствах с тремя состояниями выходов связана с использованием магистрально-модульных систем. Реализация выходов с третьим состоянием внутри кристалла ПЛИС со-

пряжена с трудностями организации их безотказной работы: необходимо строго соблюдать условие, при котором общей магистралью в каждый момент времени управляет только один элемент (драйвер). Потребность в таких линиях часто возникает при наращивании блоков ОЗУ, подключении к общим магистралям микропроцессорных устройств и т. д.

Примеры описания устройств выходов с третьим состоянием:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
-- Синтезируемое описание простого выходного буфера (драйвера)
ENTITY tristate1 IS
        PORT ( input, enable : IN std_logic;
                    output : OUT std_logic) ;
END tristate1 ;
ARCHITECTURE single_driver OF tristate1 IS
BEGIN
        output <= input WHEN enable = '1' ELSE 'Z' ;
END single_driver ;
-- Синтезируемое описание драйвера шины
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY tristate2 IS
        PORT ( input3, input2, input1, input0: IN std_logic_vector (7
DOWNTO 0);
 enable : IN std_logic_vector (3 DOWNTO 0);
 output : OUT std_logic_vector (7 DOWNTO 0) );
END tristate2 ;
ARCHITECTURE multiple_drivers of tristate2 IS
BEGIN
        output <= input3 WHEN enable(3) = '1' ELSE "ZZZZZZZZ" ;
        output <= input2 WHEN enable(2) = '1' ELSE "ZZZZZZZZ" ;
        output <= input1 WHEN enable(1) = '1' ELSE "ZZZZZZZZ" ;
        output <= input0 WHEN enable(0) = '1' ELSE "ZZZZZZZZ" ;
END multiple_drivers;
-- Синтезируемое описание шинного формирователя
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY bidir IS
 PORT ( input_val, enable: IN std_logic;
 output_val : OUT std_logic;
```

```
 bidir_port : INOUT std_logic) ;
END bidir ;
ARCHITECTURE tri_state OF bidir IS
BEGIN
        bidir_port <= input_val WHEN enable = '1' ELSE 'Z' ;
        output_val <= bidir_port;
END tri_state;
```

2.8. Пример описания ОЗУ

Для описания адресной памяти с произвольным доступом целесообразно применять общепринятый шаблон, правильно распознаваемый программами синтеза и размещаемый на ПЛИС во встроенных блоках памяти.

В примере описан блок памяти, состоящий из восьми 16-разрядных слов. Запись в память осуществляется с учетом сигналов разрешения байт BE:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
ENTITY MEM_8_16_BE IS
 PORT (
   A : IN std_logic_vector(2 DOWNTO 0);
   BE: IN std_logic_vector(1 DOWNTO 0);
   WE: IN std_logic;
   CLK: IN std_logic;
   DIN: IN std_logic_vector(15 DOWNTO 0);
   DOUT: OUT std_logic_vector(15 DOWNTO 0)
   );
END MEM_8_16_BE;
ARCHITECTURE BEHAV OF MEM_8_16_BE IS
TYPE MEM_8_16 IS ARRAY (0 TO 7) OF std_logic_vector(7
DOWNTO 0);
SIGNAL RAM0: MEM_8_16;
SIGNAL RAM1: MEM_8_16;
SIGNAL D:std_logic_vector(15 DOWNTO 0);
BEGIN
OUT0: PROCESS (CLK,WE,A,BE)
```

```
BEGIN
   IF (CLK'EVENT AND CLK='1') THEN
       IF ((WE = '1' OR WE = 'H') AND BE(0)='0') THEN
       RAM0(conv_integer(unsigned(A)))(7 DOWNTO 0) <=
DIN(7 DOWNTO 0);
          END IF;
   END IF;
END PROCESS OUT0;
DOUT(7 DOWNTO 0) <= RAM0(conv_integer(unsigned(A)))(7
DOWNTO 0);
OUT1: PROCESS (CLK,WE,A,BE)
BEGIN
     IF (CLK'EVENT AND CLK='1') THEN
        IF ((WE = '1' OR WE = 'H') AND BE(1)='0') THEN
        RAM1(conv_integer(unsigned(A)))(7 DOWNTO 0) <= DIN(15
DOWNTO 8);
          END IF;
  END IF;
END PROCESS OUT1;
DOUT(15 DOWNTO 8) <= RAM1(conv_integer(unsigned(A)))(7
DOWNTO 0);
END behav;
```

2.9. Пример описания автоматов

При проектировании цифровых устройств часто возникает необходимость в реализации цифровых автоматов. Методология проектирования устройств такого типа хорошо формализована.

Рассмотрим реализацию цифровых автоматов Мура и Мили для устройства управления, алгоритм работы которого показан на рис. 12. Как известно, в автоматах Мура выходной сигнал зависит только от текущего состояния и, в отличие от автомата Мили, не зависит от входного сигнала.

Пример реализации устройства управления в виде автомата Мура с синхронными входами:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY control_unit IS
```

```
PORT(        U : IN std_logic_vector ( 3 DOWNTO 1 );
        clk : IN std_logic;
        rst : IN std_logic;
        V : OUT std_logic_vector ( 3 DOWNTO 1 ) );
```

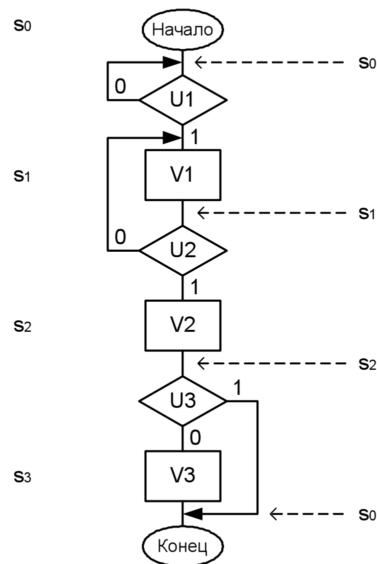

Рис. 12. Алгоритм работы устройства управления и примеры кодирования состояний для автоматов Мура и Мили

```
ARCHITECTURE moore OF control_unit IS
   TYPE STATE_TYPE IS (s0,s1,s2,s3);
   SIGNAL current_state : STATE_TYPE;
BEGIN
    clocked_proc : PROCESS (clk, rst)
    BEGIN
    IF (rst = '0') THEN
          current_state <= s0;
    ELSIF (clk'EVENT AND clk = '1') THEN
        CASE current_state IS
    WHEN s0 =>
      V(3 downto 1) <= (OTHERS => '0');
     IF (U(1)='1') THEN current_state <= s1;
```

```
      ELSE current_state <= s0; END IF;
    WHEN s1 =>
       V<= "001";
      IF (U(2) = '1') THEN current_state <= s2;
      ELSE current_state <= s1; END IF;
    WHEN s2 =>
      V <= "010";
     IF (U(3) = '0') THEN current_state <= s3;
     ELSE current_state <= s0; END IF;
     WHEN s3 =>
       V <= "100";
       current_state <= s0;
    WHEN OTHERS =>
       current_state <= s0;
     END CASE;
   END IF;
  END PROCESS clocked_proc;
END moore;
```

Пример реализации устройства управления в виде автомата Мили с синхронными входами и выходами:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY control_unit IS
     PORT(      U : IN std_logic_vector ( 3 DOWNTO 1 );
          clk : IN std_logic;
          rst : IN std_logic;
          V : OUT std_logic_vector ( 3 DOWNTO 1 ) );
ARCHITECTURE mielie OF control_unit IS
    TYPE STATE_TYPE IS (s0,s1,s2,s3);
    SIGNAL current_state : STATE_TYPE;
BEGIN
    clocked_proc : PROCESS (clk, rst)
BEGIN
    IF (rst = '0') THEN
     current_state <= s0;
   ELSIF (clk'EVENT AND clk = '1') THEN
         CASE current_state IS
           WHEN s0 =>
              IF (U(1)='1') THEN
```

```
                    V<="001";
                current_state <= s1;
  ELSIF (U(1) = '0') THEN
       V(3 DOWNTO 1) <= (OTHERS => '0');
       current_state <= s0;
         ELSE current_state <= s0; END IF;
      WHEN s1 =>
         IF (U(2) = '1') THEN
          V <= "010";
        current_state <= s2;
         ELSE current_state <= s1; END IF;
      WHEN s2 =>
         IF (U(3) = '1') THEN
          V <= "000";
       current_state <= s0;
         ELSIF (U(3) = '0') THEN
          V <= "100";
       current_state <= s0;
         ELSE current_state <= s2; END IF;
              WHEN OTHERS => current_state <= s0;
      END CASE;
     END IF;
   END PROCESS clocked_proc;
END mielie;
```

3. СИСТЕМЫ АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ ЦИФРОВЫХ УСТРОЙСТВ С ИСПОЛЬЗОВАНИЕМ ПЛИС

3.1. Процесс проектирования цифровых устройств с использованием ПЛИС

Проектирование цифровых устройств представляет собой итерационный процесс, основанный на принципах функциональной декомпозиции. Проектирование традиционно разделяют на этапы: системный, структурно-алгоритмический, функционально-логический, конструкторско-технологический.

На системном этапе весь проект разбивается на части, определяются их назначение и взаимосвязь, принимается решение о способах

реализации частей. Решение об использовании ПЛИС, принятое на системном этапе, позволяет выполнять конструкторско-технологическое проектирование модуля верхнего уровня параллельно с выполнением других этапов. Например, при проектировании контроллера шины принимается решение о реализации логики взаимодействия с шиной на основе ПЛИС. Это позволяет приступить к конструированию печатной платы сразу после определения номенклатуры ПЛИС и множеств входных и выходных сигналов.

Проектирование устройств на основе ПЛИС выполняется с применением специализированных САПР, в большой степени диктующих методику и средства разработки, а также элементную базу. При этом используются как визуальные средства проектирования, обладающие хорошей наглядностью, так и средства командного управления процессом проектирования, способствующие большей автоматизации. Проектирование с помощью таких САПР заключается в последовательном использовании предоставляемых программных средств. В терминологии САПР такой процесс называется *маршрутом проектирования*.

Структурно-алгоритмический и функционально-логический этапы проектирования устройств на основе ПЛИС базируются на итерационном вводе и верификации описаний параллельно функционирующих процессов, каждый из которых реализует заданный алгоритм.

Современные САПР поддерживают несколько способов описания устройства:

• с использованием языков описания аппаратных средств (VHDL, Verilog, AHDL и др.) и специализированного текстового редактора;

• схемотехнический способ описания с помощью программы визуального проектирования, позволяющей разработчику помещать на рабочую область функциональные блоки и производить их соединение. По окончании визуального проектирования схема преобразуется в языковое описание;

• графическое представление цифровых автоматов в специализированном редакторе, обеспечивающем преобразование полученного графического представления в языковое описание;

• описание комбинационной логики с помощью таблиц истинности, карт Карно, функций алгебры логики.

Конструкторско-технологический этап проектирования устройств с использованием ПЛИС разделяется на связанные подзадачи, схожие с подзадачами, решаемыми при схемно-топологическом проектировании на основе заказных ИС.

Данный этап включает следующие подзадачи (заметим, что англоязычная терминология в некотоных случаях не соответствует устоявшейся русской терминологии) [5].

• Синтез (Synthesis) — отображение схемы в базис логических ресурсов ПЛИС. Цель синтеза — преобразование исходного схемотехнического или высокоуровневого описания устройства в описание, оптимально реализуемое на выбранной ПЛИС, а также пригодное для дальнейшего размещения и трассировки. На стадии синтеза и после нее используются различные методы оптимизации описания, направленные на достижение наилучших результатов с точки зрения минимума требуемых ресурсов кристалла, максимума частоты синхросигнала, минимума потребляемой мощности. Например, на стадии синтеза принимается решение о способе кодирования состояний цифровых автоматов: кодирование One-Hot обеспечивает наибольшее быстродействие, в то время как иные способы (код Грея, двоичное кодирование) требуют меньших ресурсов.

• Глобальное размещение (Mapping) — назначение частям схемы макрообластей ПЛИС, представляющих собой группы соседних логических блоков, макроячеек и блоков ввода/вывода. Цель глобального размещения — создание наилучших условий для локального размещения и трассировки. Для достижения этой цели используется информация о соответствии устройства внешних выводов сигналам, существенно влияющая на назначение свободных областей ПЛИС частям схемы. Как правило, назначение логических ресурсов кристалла макрообластям производят с избыточностью, облегчающей последующую трассировку.

• Локальное размещение (Placement) — детальное назначение логических ресурсов макрообластей, выбранных на стадии глобального размещения, частям схемы. При этом преследуются цели равномерного заполнения макрообластей элементами и трассами, минимизации суммарной длины линий связи и др. Основная цель локального размещения — создание наилучших условий для трассировки.

• Трассировка (Routing) — определение связей между логическими блоками, макроячейками и блоками ввода/вывода в виде коммутированных участков трасс. На данной стадии преследуются цели выбора трасс, обеспечивающих заданное время распространения сигнала; минимизации суммарного количества программируемых точек связи; минимизации времени распространения сигнала по самой длинной линии связи. По результатам трассировки могут быть определены временные параметры полученного варианта устройства и выполнено их сравнение с заданными ограничениями.

Процесс проектирования является итерационным. После выполнения каждой подзадачи производится верификация полученного описания, для чего применяются различные средства моделирования и анализа. В современных САПР для моделирования используются следующие виды описаний:

• исходное поведенческое описание;

• описание уровня регистровых передач (RTL-описание);

• технологическое описание после синтеза;

• описание на вентильном уровне;

• технологическое описание с учетом результатов размещения;

• технологическое описание с учетом результатов трассировки.

Особое внимание разработчиков уделяется описанию тестовых воздействий, так как грамотно составленный тест позволяет выявить большинство ошибок. Описание тестов может быть выполнено несколькими способами, в том числе следующими:

• Описание генератора тестовых воздействий, не подлежащего дальнейшему синтезу. Для моделирования этим способом используется пара взаимосвязанных описаний: описание генератора и описание целевого устройства. На выходных портах целевого устройства в процессе моделирования определяется оклик на тестовое воздействие. Этот способ является универсальным — описанное на языке VHLD или Verilog тестирующее устройство-генератор может успешно применяться в любом моделирующем пакете. Кроме того, при описании теста на языках VHDL, Verilog могут быть использованы широкие алгоритмические возможности языков: циклы, ожидания, ловушки, сообщения, генерации и т. д.

• Описание тестовых воздействий в специальном графическом редакторе. Такой способ подходит для простых тестовых воздействий.

После выполнения трассировки и верификации результатов автоматически может быть генерирован файл с конфигурационной последовательностью, содержащий информацию о коммутации и функциональности всех ресурсов кристалла. На заключительном этапе маршрута проектирования выполняются программирование ПЛИС и последующая внутрисхемная верификация устройства (проверка работоспособности на макетной ПЛИС). При этом разработчик может использовать дополнительное оборудование (осциллографы, логические анализаторы, генераторы сигналов) или воспользоваться специализированными, встраиваемыми в ПЛИС, логическими анализаторами.

3.2. САПР Xilinx ISE

Маршрут проектирования в САПР Xilinx ISE

Фирма Xilinx является мировым лидером в производстве ПЛИС: ею производится около 50 % современных ПЛИС. Помимо самих микросхем с программируемой структурой Xilinx также предоставляет программные средства проектирования, такие как система сквозного проектирования ISE. Эта САПР автоматизирует все этапы проектирования с использованием ПЛИС, от ввода описания устройств до его внутрисхемной верификации.

Фирма Xilinx выпускает несколько версий САПР ISE:

• версия Foundation с неограниченной номенклатурой используемых ПЛИС и средствами внутрисхемной верификации;

• свободно распространяемая версия Web Edition с ограниченной номенклатурой используемых ПЛИС и упрощенными версиями некоторых модулей (ограничение касается кристаллов с большой емкостью, модуля симуляции и поддержки операционной системы Sun Solaris и 64-разрядных операционных систем). Версия доступна для загрузки по адресу: http://www.xilinx.com/ise/logic_design_prod/webpack.htm.

Рассмотрим более подробно версию Xilinx ISE 9.1 Web Edition для операционной системы Windows XP. В ее состав входят следующие модули.

• Навигатор проекта (Project Navigator) — программа, интегрирующая используемые в маршруте проектирования модули.

• Схемотехнический редактор (Schematic Editor), обеспечивающий графический ввод схем, создание новых примитивов, использование библиотечных примитивов.

• Редактор ввода на языках VHDL и Verilog (HDL Editor), обеспечивающий ввод языковых описаний.

• Графический редактор цифровых автоматов (State Cad) — средство визуального проектирования автоматов, их моделирования и преобразования графического описания в описание на языке VHDL или Verilog.

• Генератор устройств (CORE Generator) — программа-мастер для интерактивной настройки описаний часто используемых компонентов (в терминологии ПЛИС — ядер).

• Редакторы ограничений (Constraints Editor, Pinout and Area Constraints Editor) — программы для создания и редактирования файла ограничений проекта (User Constraints File, UCF), используемого при синтезе, размещении и трассировке.

• Синтезатор (XST Tool) — программа синтеза низкоуровневого описания устройств.

• Программы визуализации низкоуровневых описаний (RTL Viewer, Technology Viewer), представляющие низкоуровневые описания в схемотехническом виде.

• Программа функционального и временнóго моделирования (ISE Simulator Lite), позволяющая также в визуальном режиме редактировать тестовые воздействия.

• Программа анализа временных параметров (Static Timing Analyzer), определяющая времена распространения сигналов и сравнивающая их с заданными ограничениями.

• Программы автоматического размещения и трассировки ПЛИС (MAP Tool, PAR Tool), учитывающие заданные временные ограничения.

• Программы ручного размещения и оптимизации проекта (Floor Planner, FPGA Editor)

• Программа для анализа рассеиваемой мощности (XPower).

• Программы загрузки конфигурационной последовательности в ПЛИС FPGA и программирования ПЛИС CPLD и ППЗУ (iMPACT).

Совместно с САПР ISE Web Edition фирма Xilinx предоставляет программу моделирования ModelSim XE III Starter фирмы

MentorGraphics с ограниченной лицензией. Более подробное описание указанных модулей будет приведено ниже.

На рис. 13 показан обобщенный маршрут проектирования, используемый в САПР Xilinx ISE.

Проектирование начинается с создания проекта в программе Project Navigator. При этом указываются ключевые параметры проекта (номенклатура ПЛИС, тип корпуса, способ описания модуля верхнего уровня), выбираются одна из доступных программ синтеза и программа моделирования, выбирается предпочитаемый язык описания (Verilog и VHDL).

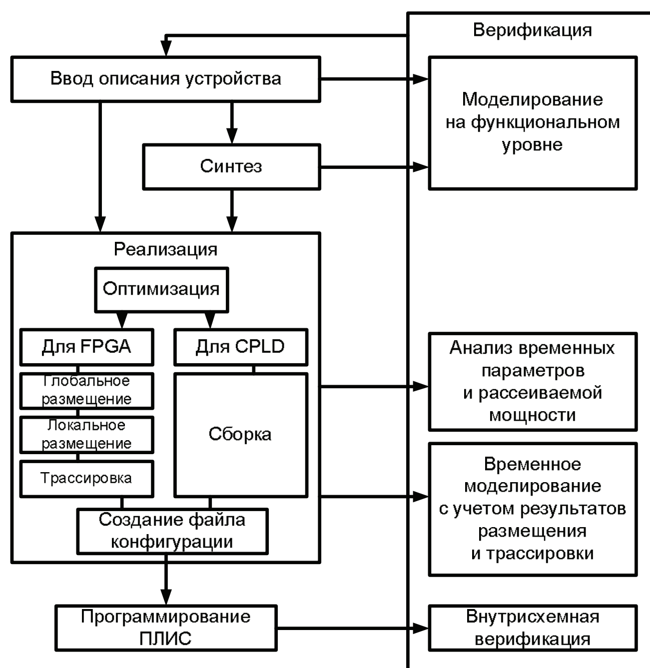

Рис. 13. Маршрут проектирования в САПР Xilinx ISE

Следующей стадией является ввод описаний устройств, объединенных в дерево проекта. Вершиной дерева является модуль верхнего уровня, объединяющий остальные функциональные части проекта. Модуль верхнего уровня может быть описан с помо-

щью языка VHDL или Verilog, а также схемотехнического редактора. На данной стадии используются модули: Schematic Editor, HDL Editor, State Cad, CORE Generator. Помимо структуры устройств описываются тестовые воздействия и ограничения реализации. Описание тестов в САПР Xilinx ISE может быть выполнено с помощью разработки генератора тестовых воздействий или графического редактора тестовых воздействий ISE Simulator Lite.

В САПР Xilinx ISE предусматривается несколько уровней моделирования, выполняемых в программе ModelSim XE III Starter:

• функциональное моделирование RTL-описания, технологического описания после синтеза (Behavioral Simulation) и описания на вентильном уровне (Post Translate Simulation);

• временное моделирование описания после размещения (Post-Map Simulation) и моделирование после трассировки (Post-Route Simulation).

При использовании модуля ISE Simulator Lite доступны только моделирование поведенческого описания (Behavioral Simulation) и описания после размещения и трассировки (Post Translate Simulation).

В целях верификации может быть выполнен анализ временных параметров устройства с помощью модуля Static Timing Analyzer и анализ рассеиваемой мощности с помощью программы XPower.

Для описания ограничений в САПР Xilinx ISE могут быть использованы модули: Constraints Editor для описания ограничений ввода/вывода и временных параметров, а также PACE для ограничения размещения.

После стадии ввода описания выполняется автоматический синтез низкоуровневого описания, результаты которого могут быть просмотрены с помощью модулей RTL Viewer и Technology Viewer.

Далее выполняются автоматизированные стадии размещения и трассировки, полученные результаты могут быть проанализированы и вручную отредактированы с помощью модулей ручного размещения и оптимизации проекта Floor Planner и FPGA Editor.

Маршрут реализации проекта на основе ПЛИС типа CPLD несколько отличается от маршрута реализации FPGA. Так, этапы размещения и трассировки реализованы в виде одного этапа — сборки (Fitting).

После стадии трассировки генерируется файл конфигурации ПЛИС, который может быть переформатирован в файл содержимого конфигурационного ППЗУ. На этой стадии разработчик задает последовательность начальной загрузки конфигурации в ПЛИС и программирует с помощью модуля iMPACT все необходимые для этого устройства (ППЗУ или ПЛИС).

Навигатор проекта Project Navigator

Навигатор проекта реализован с помощью многооконного интерфейса, в котором пользователь может по своему усмотрению настраивать состав и набор видимых окон. На рис. 14 представлен вид навигатора проекта, выбираемый по умолчанию. Помимо основного меню и кнопок быстрого запуска, активизирующихся в соответствии со стадией проектирования, доступны окна, визуализирующие иерархию описаний проекта (окно Sources), окно с указанием результатов выполнения стадий проектирования (окно Project), окно с текстовой информацией и средствами командного управления (окно Transcript).

В центральной части окна выделено рабочее поле для размещения интегрируемых в навигатор интерфейсов модулей. Большинство из перечисленных выше модулей могут быть размещены в рабочем поле навигатора проекта или использованы независимо в виде отдельного приложения. При любом способе можно задействовать стандартные возможности интерфейса со свободной навигацией: выбор инструментов в главном меню и на панели быстрого запуска, контекстное меню и т. д.

Схемотехнический редактор Schematic Editor

Модуль Schematic Editor позволяет создавать, просматривать, модифицировать и соединять компоненты устройства, отображаемые в символическом виде. Такой способ традиционно используется при схемотехническом описании устройств, однако имеет свои недостатки: схема достаточно трудно понимается и модифицируется; сложные устройства занимают большое рабочее пространство; графический способ описания заметно медленнее текстового; поиск ошибок затруднителен.

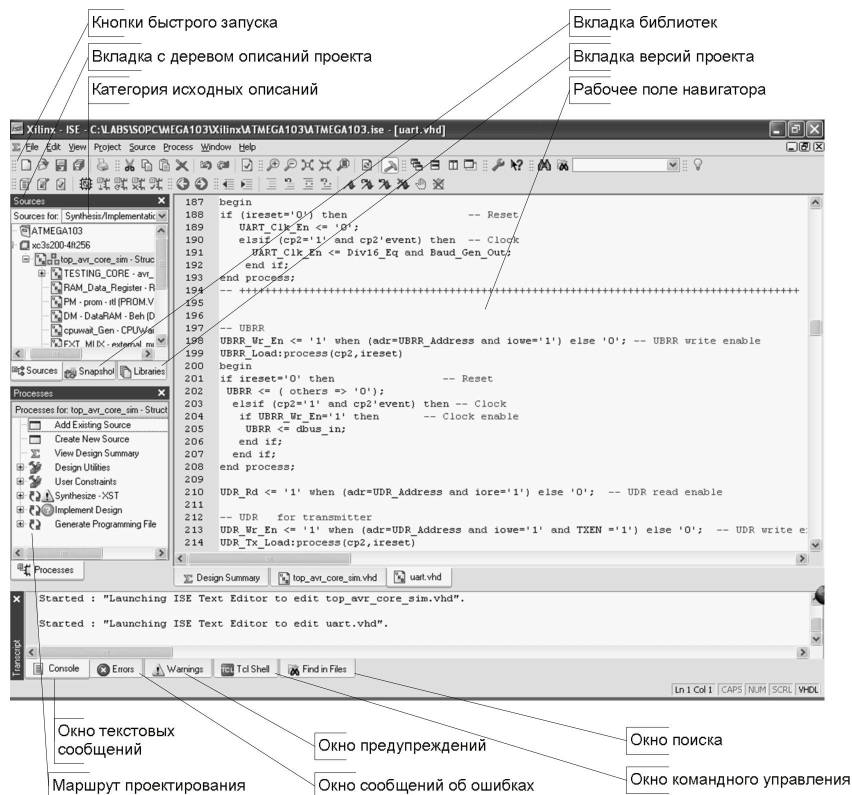

Рис. 14. Навигатор проекта Xilinx ISE

С помощью модуля Schematic Editor целесообразно описывать небольшие части устройства, а также блоки, содержащие библиотечные схемотехнические компоненты, реализация которых хорошо известна разработчику. Целесообразно использовать Schematic Editor для описания модуля верхнего уровня (рис. 15).

Описание устройства производится на рабочем поле. Для выбора компонентов используется вкладка Symbols окна Sources. Соединение компонентов выполняется с помощью цепей, шин и разветвителей. Для создания цепи или шины необходимо над рабочим полем вызвать контекстное меню с помощью правой кнопки мыши, далее в контекстном меню выбрать пункт Add и подпункт

Wire. Если порт компонента является шиной (Bus), то подключенная к нему цепь будет автоматически преобразована в шину.

Для подключения цепи или группы цепей к существующей шине используется разветвитель Bus Tap. Для его добавления в контекстном меню следует выбрать пункты Add и Bus Tap, после чего необходимо поместить разветвитель на шине. Указание на соответствие цепей в шине цепям, подведенным к разветвителю, выполняется с помощью имен. Например, если шина носит имя XLXN_1(15:0), то для выбора старшей цепи, подведенной к разветвителю, следует дать название XLXN_1(15).

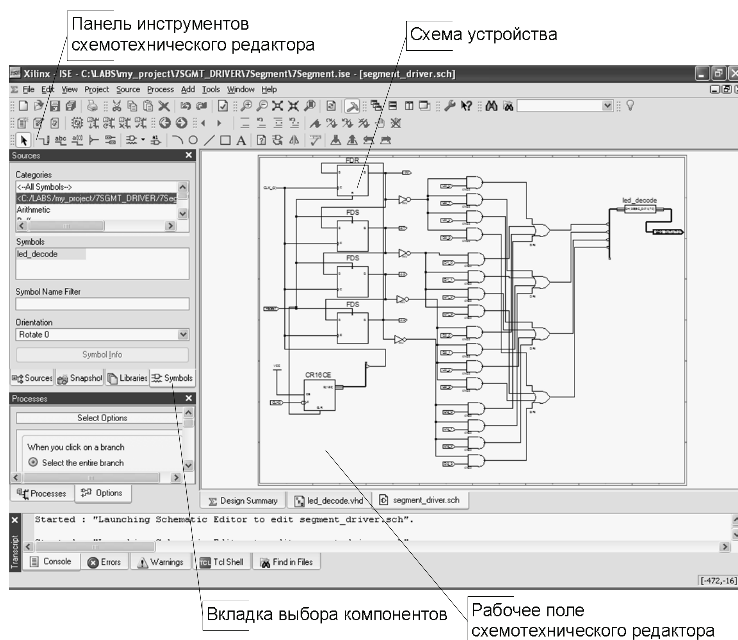

Рис. 15. Модуль Schematic Editor

Порты схемы подключаются к цепям с помощью компонента I/O Marker. Для этого в контекстном меню необходимо выбрать пункты Add и I/O Marker. Если в процессе отладки требуется определить состояние внутренних сигналов устройства, их также необходимо подключить к портам вывода.

Редактор HDL Editor

Модуль HDL Editor (рис. 16) дает возможность создавать и редактировать текстовые описания устройств. Он обеспечивает подсветку синтаксиса описаний на языках VHDL и Verilog, позволяет устанавливать точки останова, используемые при моделировании, а также поддерживает поиск строк в RTL- и технологическом описаниях.

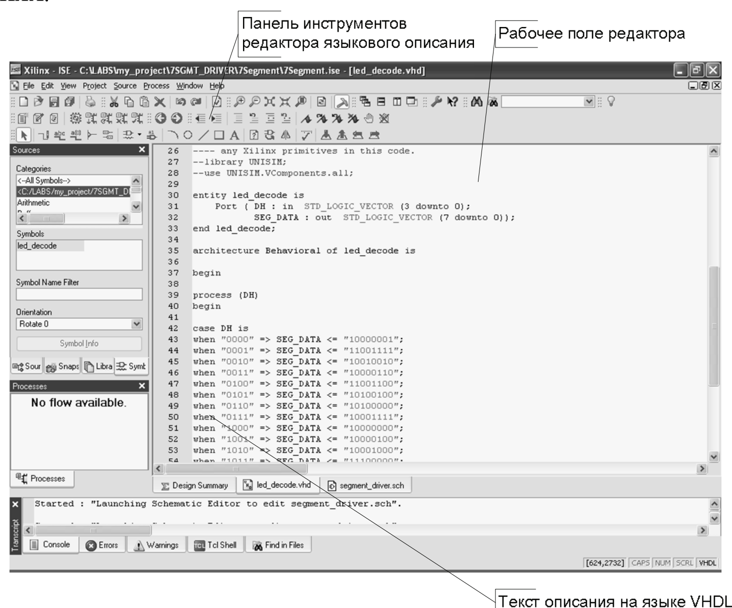

Рис. 16. Внешний вид модуля HDL Editor

При активизации редактора на рабочем поле навигатора проекта становится активной панель инструментов, содержащая кнопки: увеличение и уменьшение отступа, комментирование выделенных фрагментов, создание точек останова.

Графический редактор цифровых автоматов State Cad

Модуль State Cad является средством, позволяющим визуально проектировать описания цифровых автоматов. Представление авто-

мата в виде диаграммы состояний обладает наглядностью, его легко модифицировать, проще искать ошибки. В связи с этим такой способ проектирования автоматов является предпочтительным.

Для описания автомата в модуле State Cad необходимо поместить на рабочее поле модуля графические представления состояний и соединить их дугами — переходами (рис. 17). Для каждого состояния и перехода можно задать состояние выходных сигналов автомата. Помимо описания автомата возможно использовать параллельно работающие стандартные компоненты: счетчики, регистры, дешифраторы и т. д. Это позволяет визуально проектировать достаточно сложные устройства.

Модуль State Cad обеспечивает моделирование работы автомата и в наглядном виде представляет результаты. После разработки устройства может быть автоматически генерировано синтезируемое описание на языке VHDL или Verilog.

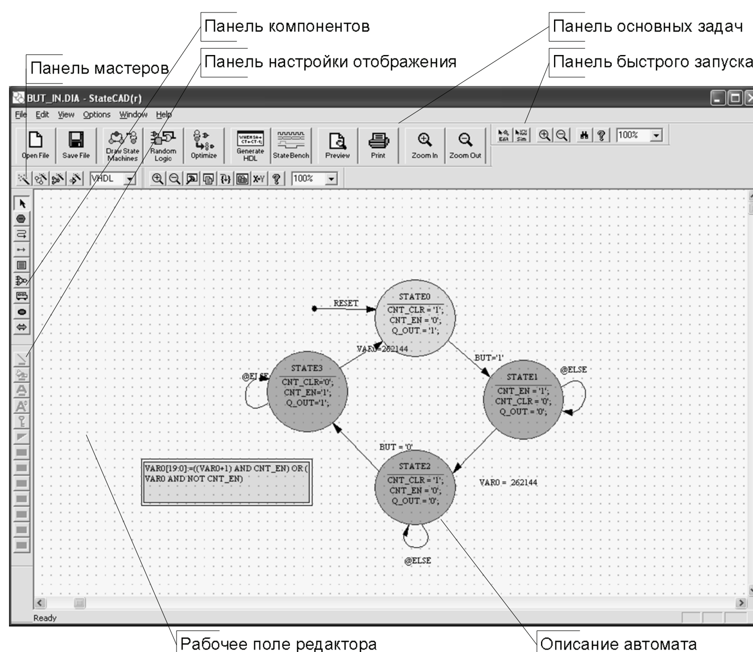

Рис. 17. Модуль State Cad

Окно редактора цифровых автоматов содержит панели с необходимыми кнопками: панель мастеров, панель быстрого запуска, панель настройки изображения, панель компонентов.

Генератор настраиваемых компонентов CORE Generator

Модуль CORE Generator (рис. 18) позволяет включать в проект готовые описания параметризированных объектов интеллектуальной собственности, оптимизированных для размещения на ПЛИС типа FPGA.

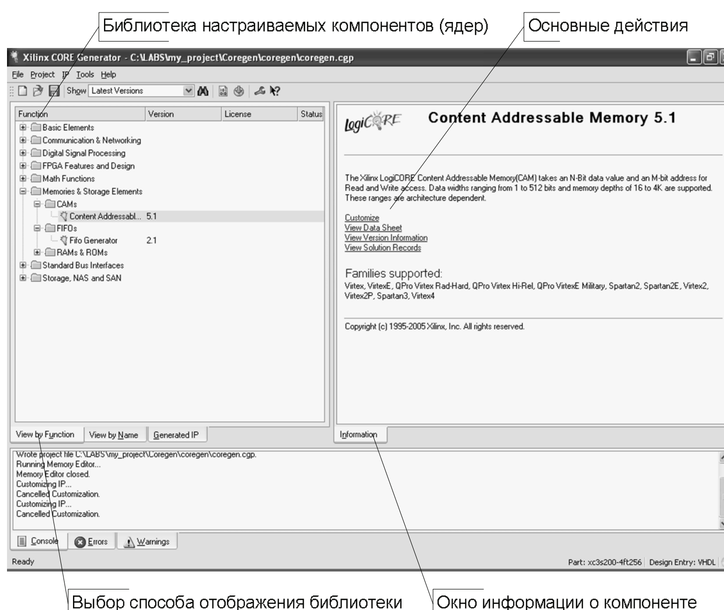

Рис. 18. Модуль CORE Generator

В модуле доступны описания таких устройств, как адресная память, буферы FIFO, ассоциативные ЗУ, фильтры, интерфейсная логика (PCI, PCI-X), устройства обработки чисел с плавающей запятой и т. д.

Для использования IP-ядер необходимо выбрать один из доступных компонентов в библиотеке компонентов и пройти процедуру параметризации, выбрав пункт Customize.

Программа функционального и временного моделирования ISE Simulator Lite

Модуль ISE Simulator Lite может быть использован в качестве редактора тестовых воздействий (Graphical Testbench Editor View) и программы функционального и временного моделирования (ISE Simulator).

Для создания нового теста необходимо в меню Project выбрать пункт New Source. В появившемся окне следует выбрать пункт Test Bench Wave Form. После указания имени файла и ассоциации с тестируемым описанием проекта разработчик может задать дополнительные параметры: интервал моделирования, параметры сигнала синхронизации, шаг моделирования и др. В результате будут созданы тестовые воздействия для входных портов (рис. 19). Разработчик может изменить воздействие для любого порта с помощью мыши и (или) панели быстрого запуска.

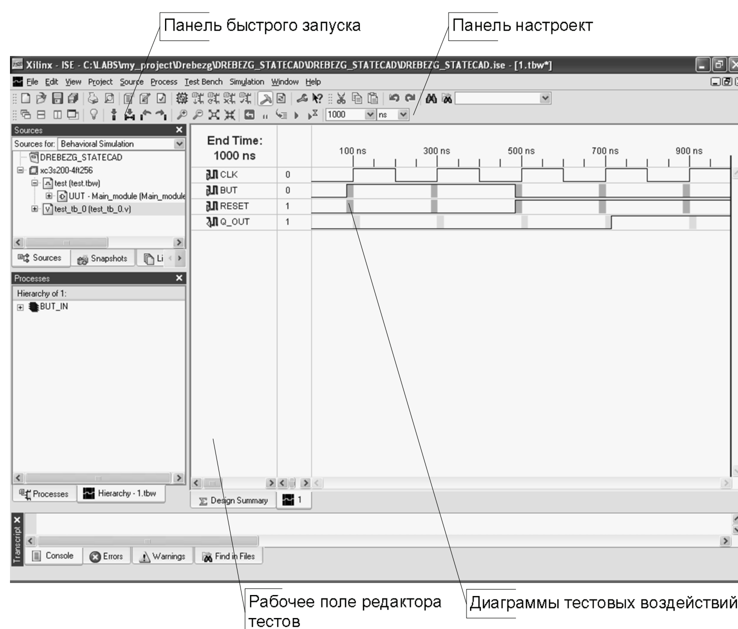

Рис. 19. Модуль ISE Simulator Lite в режиме редактора тестовых воздействий

В режиме моделирования (рис. 20) программа ISE Simulator Lite позволяет выполнить моделирование описаний на языках VHDL и Verilog, визуализирует полученные диаграммы откликов устройства на тестовые воздействия. Состояние шин может быть представлено в шестнадцатеричном, десятичном (со знаком или без знака), двоичном или ASCII формате.

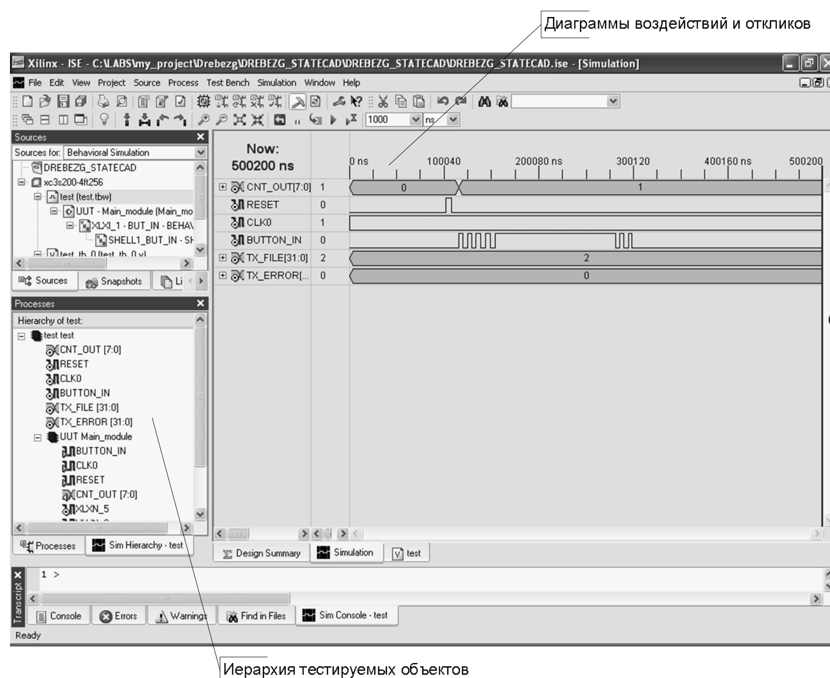

Рис. 20. Модуль ISE Simulator Lite

Редактор ограничений Constraints Editor

Ограничения представляют собой текстовые инструкции программам размещения и трассировки. Они могут влиять на размещение частей устройства на кристалле ПЛИС, определяют имена цепей и начальное содержимое блочной и распределенной памяти, определяют направление и временные параметры цепей. В САПР Xilinx ISE ограничения сохраняются в текстовом формате в файле *.ucf.

Модуль Constraints Editor (рис. 21) обеспечивает:

• задание глобальных временных ограничений;

• задание временных ограничений для отдельных портов или групп портов;

• создание групп компонентов и контрольных точек, для которых описываются ограничения;

• создание ограничений на начальное состояние динамических триггеров, триггеров-защелок, памяти, регистров сдвига.

Модуль Constraints Editor предоставляет графический способ создания и редактирования файла ограничений проекта. Он содержит окно ограничений, в котором перечисляются названия сигналов и задаваемые для них условия. На вкладке Misc задаются начальные состояния триггеров и памяти.

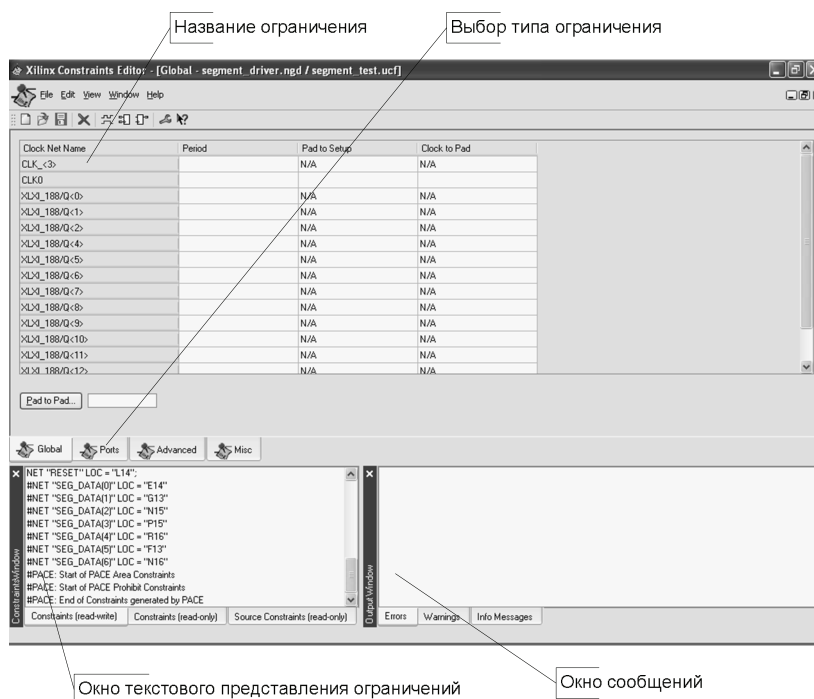

Рис. 21. Модуль Constraints Editor

При реализации сложных быстродействующих устройств особое внимание уделяется заданию временных ограничений для цепей. Для задания временных ограничений должны быть определены сигнал, используемый для синхронизации, и его период. После этого для каждой цепи можно указать два типа временных ограничений: на время прохождения сигнала от контакта до входа (Pad to Setup) и время прохождения сигнала от входа до контакта (Clock to Pad).

Время прохождения сигнала от контакта до входа определяется как максимально допустимый промежуток времени между событием изменения сигнала на контакте ПЛИС до момента обнаружения этого события на входе первого синхронного элемента памяти (рис. 22, *а*), требуемый для прохождения сигнала через входные буферные элементы, цепи и комбинационную логику.

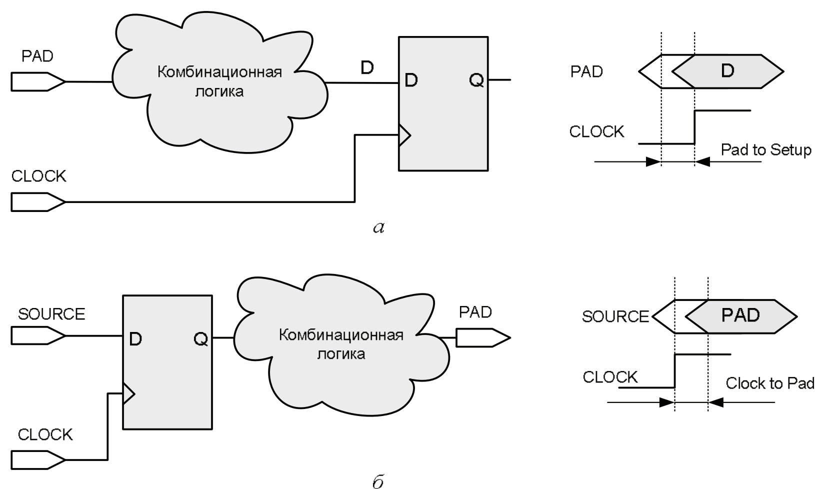

Рис. 22. Время прохождения сигнала:
а — от контакта до входа (Pad to Setup); *б* — от входа до контакта (Clock to Pad)

Время прохождения сигнала от входа до контакта определяется как максимально допустимый промежуток времени между событием изменения сигнала на входе триггера до момента обнаружения события на контакте ПЛИС, требуемый для переключения триггера (время Clock to Q), прохождения сигнала через цепи, комбинационную логику и выходной буферный элемент (рис. 21, *б*).

Программы визуализации низкоуровневых описаний RTL Viewer и Technology Viewer

Модули RTL Viewer и Technology Viewer используются для схематического представления синтезированного описания устройства, получаемого после применения синтезатора Xilinx Synthesis Technology (XST). Использование модулей RTL Viewer и Technology Viewer позволяет:

- проанализировать описание после синтеза;
- выполнить анализ критических цепей;
- выявить способы дальнейшей оптимизации описания.

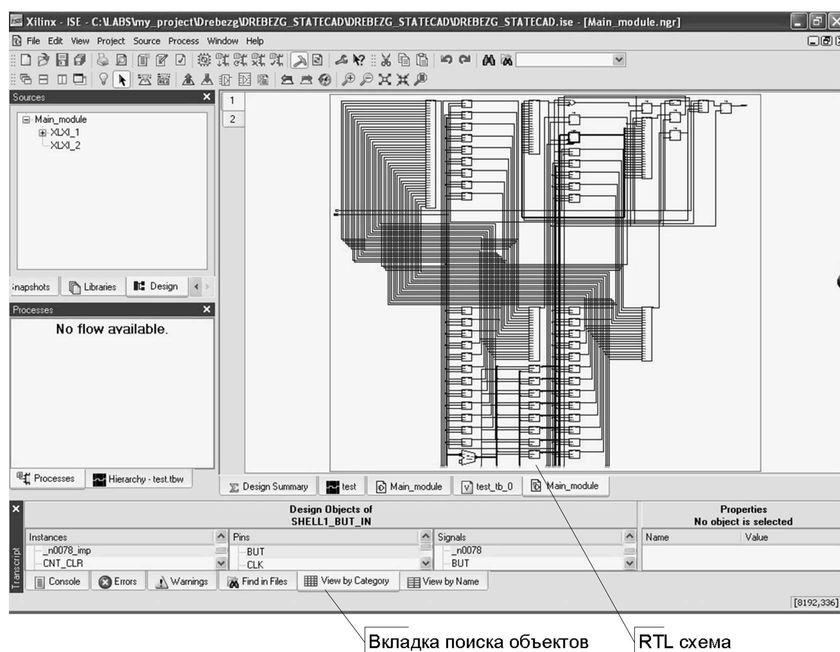

Рис. 23. Модуль RTL Viewer

Модуль RTL Viewer визуализирует описание уровня регистровых передач. Это описание является результатом синтеза, однако в нем не используются особенности конкретной ПЛИС. Технологическое описание, отображаемое в Technology Viewer, представляет

собой оптимизированное RTL-описание, представленное в логических ресурсах конкретного кристалла ПЛИС (в блоках LUT, схемах ускоренного переноса, буферах ввода/вывода и т. д.).

Внешний вид модуля RTL Viewer показан на рис. 23 (вид модуля Technology Viewer аналогичен, за исключением самой схемы). Для поиска и выделения отображаемых компонентов может быть использована вкладка поиска.

Программа ручного размещения Floor Planner

Модуль Floor Planner (рис. 24) позволяет:

• выполнять детальное размещение низкоуровневого описания устройства на кристалле ПЛИС;

• сохранять результаты размещения частей устройства в виде макросов, которые могут быть использованы в других проектах;

• просматривать и модифицировать ограничения на размещение;

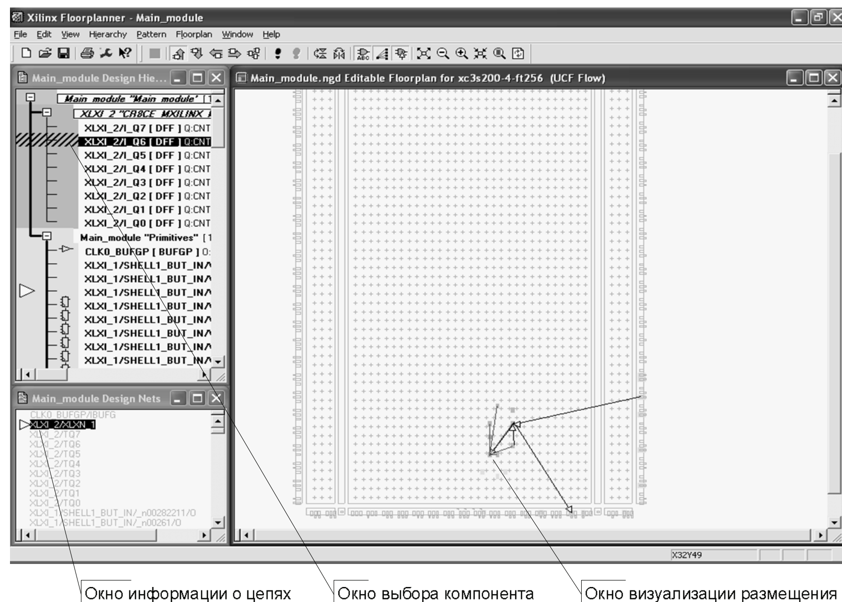

Рис. 24. Модуль Floor Planner

• выполнять поиск компонентов или цепей схемы на кристалле ПЛИС;

• визуализировать компоненты, выбранные в модуле Timing Analyzer.

Интерфейс модуля Floor Planner состоит из меню, панелей быстрого запуска, окна выбора компонентов, окна информации о цепях и окна визуализации размещения. Для размещения компонента необходимо указать его в окне выбора компонентов, после чего может быть позиционирована область в окне визуализации размещения. Одновременно с этим визуализируются связи выбранного компонента с другими размещенными компонентами и с блоками ввода/вывода, что позволяет оценить связность компонента и определить наилучший вариант расположения области.

Редактор ограничений Pinout and Area Constraints Editor

Модуль Pinout and Area Constraints Editor (PACE) может быть использован при вводе описания устройства или после синтеза низкоуровневого описания для указания ограничений на назначения контактов и выбор областей кристалла.

Назначение контактов позволяет связать порты устройства с блоками ввода/вывода и контактами микросхемы ПЛИС. Модуль PACE дает возможность для каждой цепи ввода/вывода выбрать контакт, банк и стандарт ввода/вывода. При назначении контактов для сигналов синхронизации целесообразно задействовать специальные входы GCLKx.

Модуль PACE может также быть использован в качестве программы просмотра реализации устройства для отображения большого количества структурной информации.

Интерфейсная часть модуля (рис. 25) состоит из окон выбора ресурсов, ввода ограничений, визуализации параметров и окна визуализации ресурсов ПЛИС. Для создания ограничения необходимо указать тип ресурса в окне выбора ресурсов и название ограничения в окне ограничений. После этого следует изменить параметры ограничений в окне параметров.

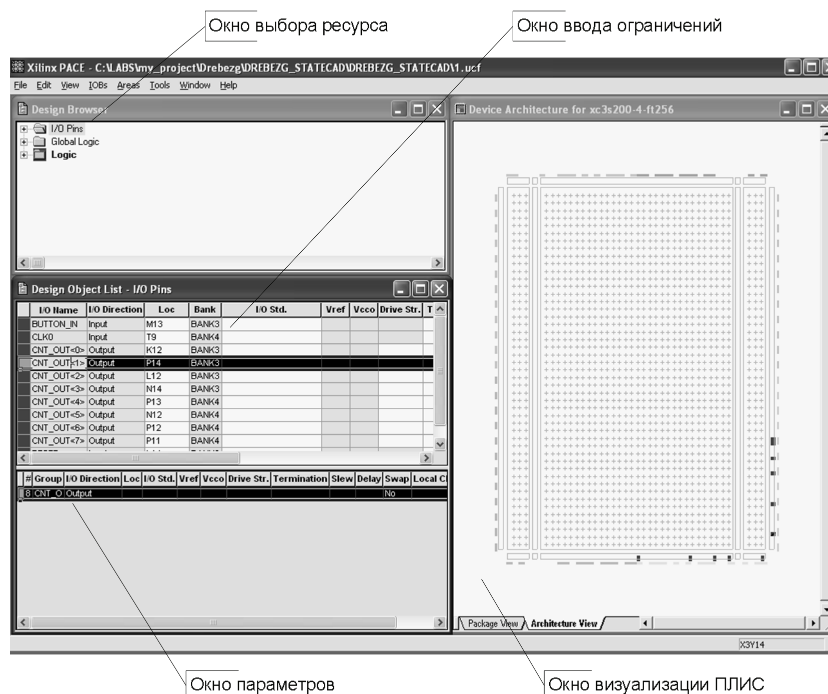

Рис. 25. Модуль PACE

Редактор ресурсов FPGA Editor

Модуль FPGA Editor (рис. 26) предназначен для визуального отображения и редактирования результатов размещения и трассировки ПЛИС типа FPGA. Модуль позволяет:

• выполнять размещение и трассировку выбранных частей устройства до применения автоматического размещения и трассировки;

• завершать процедуру размещения и трассировки в том случае, когда автоматические размещение и трассировка невозможны;

• проводить тестирование устройства благодаря соединению свободных контактов ПЛИС с внутренними цепями устройства;

• выполнять визуализацию частей устройства, выбранных в модуле Timing Analyzer;

• запускать генерацию конфигурационной последовательности и программирование ПЛИС.

Для наглядного представления логических и коммутационных ресурсов кристалла в модуле FPGA Editor предусмотрено окно визуализации ресурсов FPGA, в котором отображаются свободные и занятые коммутируемые логические блоки и трассы кристалла. Окно редактирования позволяет найти ресурс на ПЛИС и вручную изменить его свойства. Выбор команд, необходимых для редактирования результатов размещения и трассировки, может быть выполнен с помощью панели управления.

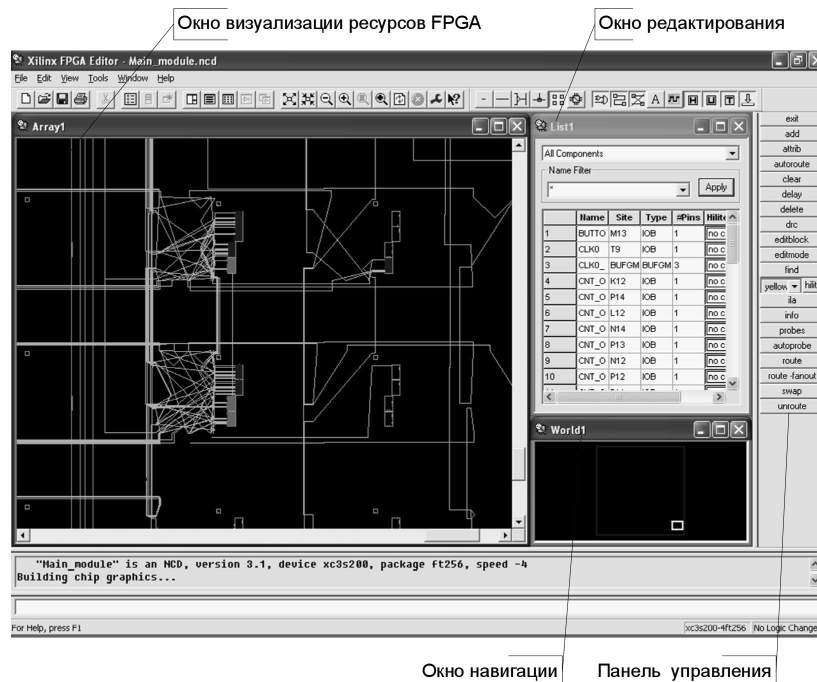

Рис. 26. Модуль FPGA Editor

Модуль анализа временных параметров Static Timing Analyzer

Модуль Static Timing Analyzer позволяет выполнить временной анализ устройств, реализуемых на основе ПЛИС серий Spartan-3A, Virtex-4 и Virtex-5.

Модуль Static Timing Analyzer обеспечивает:

- выполнение анализа статических временных параметров устройства;
- выполнение анализа временных параметров критических линий связи и линий связи с указанными ограничениями;
- выполнение анализа всех путей прохождения сигналов от контактов до входов и от входов до контактов и определяет наиболее длинный путь;
- генерацию отчета о результатах выполнения заданных ограничений;
- переход к окнам визуализации в модулях Technology Viewer and Floor Plan для просмотра выбранных путей.

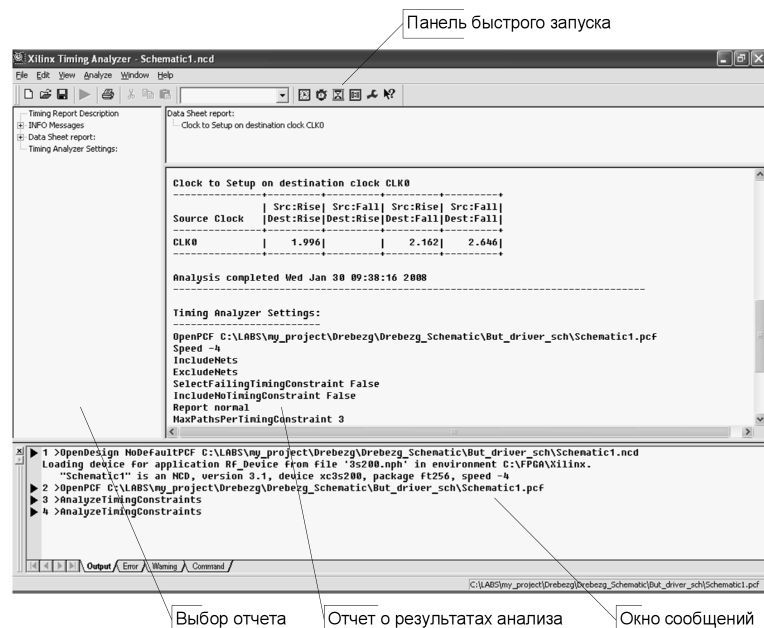

Рис. 27. Модуль Static Timing Analyzer

Для формирования отчета необходимо указать его в окне выбора отчета, затем запустить соответствующую команду на панели быстрого запуска. После формирования отчет будет показан в окне отчета о результатах анализа (рис. 27).

Модуль iMPACT

Модуль iMPACT (рис. 28) может быть использован как средство генерации конфигурационных данных, как программатор ПЛИС типа CPLD и ППЗУ или как средство конфигурации FPGA. При этом можно использовать графический интерфейс iMPACT или же обращаться к нему из командной строки.

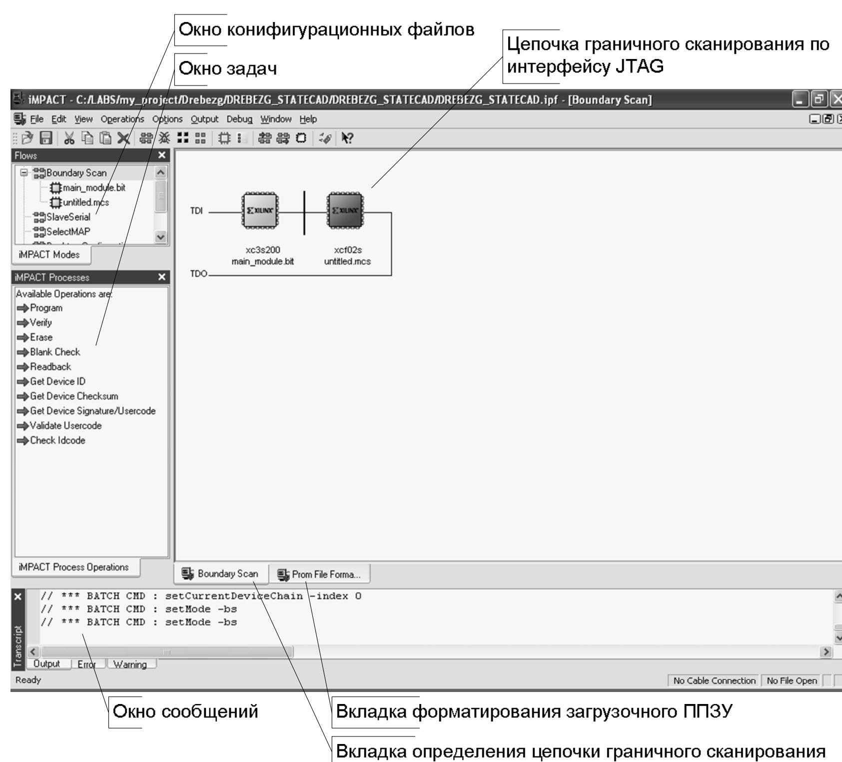

Рис. 28. Модуль iMPACT

Функции программирования и конфигурации позволяют с помощью специального оборудования выполнять запись конфигурационных данных в ПЛИС типа FPGA или CPLD, а также специализированные загрузочные ППЗУ. Программированием в этом случае принято называть процедуру записи конфигурационных данных в энергонезависимую память конфигурации (используется в CPLD и ППЗУ), а конфигурированием — запись в статическую память конфигурации (используется в FPGA). Программирование и конфигурация могут производиться в различных режимах:

• в режиме граничного сканирования по интерфейсу JTAG (для конфигурации FPGA, CPLD и ППЗУ);

• в режиме Slave Serial или SelectMAP (для прямой конфигурации FPGA без использования ППЗУ);

• в режиме Desktop Configuration для программирования CPLD и ППЗУ;

• в режиме SPI Configuration для программирования ППЗУ по интерфейсу SPI (для устройств типа M25P, M25PE, M45PE, AT45DB).

Функция генерации конфигурационных данных дает возможность создавать следующие типы файлов конфигурации: System ACE CF, PROM, SVF, STAPL и XSVF. Файл System ACE CF позволяет конфигурировать устройства типа System ACE CF, в которых процессом конфигурации FPGA управляет микропроцессор. Для таких устройств возможно автоматизированное управление конфигурирацией и переконфигурирацией.

Форматирование файлов конфигурации ППЗУ (файл PROM) позволяет:

• генерировать конфигурационные данные в виде формата, совместимого со сторонними программаторами ППЗУ;

• создавать мультизагрузочные ППЗУ для ПЛИС Virtex-5, Spartan-3A и Spartan-3E;

• соединять несколько конфигурационных файлов в один файл конфигурации ППЗУ для цепного программирования.

• сохранять несколько вариантов конфигурации одной и той же ПЛИС в ППЗУ.

Файлы в форматах SVF, STAPL и XSVF содержат конфигурационные данные и команды. Они могут быть использованы при

автоматизации программирования в режиме граничного сканирования.

Кроме этого iMPACT позволяет выполнять чтение и верификацию конфигурационного содержимого ПЛИС и ППЗУ, контроль процесса программирования устройств, исполнять файлы типа SVF и XSVF.

ЗАКЛЮЧЕНИЕ

Технология проектирования цифровых устройств с использованием программируемых логических интегральных схем стремительно развивается. Фирмы-производители включают в состав ПЛИС не только конфигурируемые логические ресурсы и память, но и более сложные устройства: интерфейсные контроллеры и микропроцессоры. Разработка и отладка комплексных систем на кристалле (System on Programmable Chip, SOPC) осуществляется с помощью специализированных САПР. Таким образом, технологии проектирования на основе ПЛИС и микропроцессорных устройств успешно сочетаются, а конечные изделия обладают многими преимуществами: высокой функциональностью и легкостью модификации, большим быстродействием, малым временем разработки и большой степенью автоматизации. Все это позволяет говорить о грядущем развитии технологий, связанных с ПЛИС и системами на кристалле, равно как и о насущной потребности в высококвалифицированных инженерах, владеющих такими технологиями.

СПИСОК ЛИТЕРАТУРЫ

1. Xilinx ISE Guide (HTML Book). Xilinx Inc. — www.xilix.com

2. Spartan-3 FPGA Family: Complete Data Sheet. Xilinx Inc.

3. XC9500 CPLD Family: Complete Data Sheet. Xilinx Inc.

4. IEEE VHDL-93 Standard 2000 Revision

5. *Угрюмов Е.П.* Цифровая схемотехника: Учеб. пособие для вузов. 2-е изд., перераб. и доп. СПб.: БХВ-Петербург, 2004. 800 с.

6. *Грушвицкий Р.И., Мурсаев А.Х., Угрюмов Е.П.* Проектирование систем на микросхемах с программируемой структурой. СПб.: БХВ-Петербург, 2006. 708 с.

7. *Сергиенко А.М.* VHDL для проектирования вычислительных устройств. К ЧП «Корнейчук», ООО «ТИД «ДС», 2003. 208 с.

8. *Зотов В.Ю.* Проектирование цифровых устройств на основе ПЛИС фирмы Xilinx в САПР WebPACK ISE. М.: Горячая линия – Телеком, 2003. 624 с.

ОГЛАВЛЕНИЕ

Предисловие 3

1. Архитектура ПЛИС 4
   1.1. Программируемые логические матрицы и программируемая матричная логика 4
   1.2. Базовые матричные кристаллы 6
   1.3. ПЛИС типа CPLD 9
   1.4. ПЛИС типа FPGA 12

2. Основы языка VHDL 18
   2.1. Назначение языка VHDL 18
   2.2. Объекты языка VHDL и их типы 20
      Базовые типы данных 20
      Атрибуты 21
      Константы 22
      Сигналы 23
      Переменные 24
      Файлы 24
      Сигналы типа STD_LOGIC и STD_LOGIC_VECTOR 24
      Операции языка VHDL 25
   2.3. Интерфейс и архитектура устройств 27
   2.4. Пакеты и библиотеки 29
   2.5. Параллельные операторы 31
      Оператор BLOCK 31
      Оператор PROCESS 32
      Оператор параллельного присваивания сигнала 33
      Процедуры и функции 34
      Оператор-ловушка ASSERT 36
      Использование компонентов 36
      Оператор GENERATE 38
   2.6. Последовательные операторы 38
      Оператор ожидания WAIT 39

Последовательный оператор-ловушка ASSERT 39
Последовательный оператор присваивания сигнала 40
Оператор присваивания переменной 41
Оператор вызова процедуры 41
Условный оператор IF 41
Оператор выбора CASE 42
Оператор цикла LOOP 43
Операторы NEXT, EXIT, RETURN, NULL 44
2.7. Пример описания устройств с тремя состояниями выходов 45
2.8. Пример описания ОЗУ 47
2.9. Пример описания автоматов 48
3. Системы автоматизированного проектирования
цифровых устройств с использованием ПЛИС 51
3.1. Процесс проектирования цифровых устройств
с использованием ПЛИС 51
3.2. САПР Xilinx ISE 55
Маршрут проектирования в САПР Xilinx ISE 55
Навигатор проекта Project Navigator 59
Схемотехнический редактор Schematic Editor 59
Редактор HDL Editor 62
Графический редактор цифровых автоматов State Cad 62
Генератор настраиваемых компонентов CORE Generator 64
Программа функционального и временного моделирования
ISE Simulator Lite 65
Редактор ограничений Constraints Editor 66
Программы визуализации низкоуровневых описаний
RTL Viewer и Technology Viewer 69
Программа ручного размещения Floor Planner 70
Редактор ограничений Pinout and Area Constraints Editor 71
Редактор ресурсов FPGA Editor 72
Модуль анализа временных параметров Static Timing Analyzer 74
Модуль iMPACT 75
Заключение 77
Список литературы 78