

Языки структурно-функционального описания цифровых систем.

Конспект лекций. Рабочая версия.

Кустарев П.В.
Кафедра Вычислительной Техники
СПбГИТМО (ТУ)
Санкт-Петербург 2006г.

ОГЛАВЛЕНИЕ.

1 ОБЩИЕ ПРИНЦИПЫ СТРУКТУРНО-ФУНКЦИОНАЛЬНОГО ОПИСАНИЯ АППАРАТУРЫ.....	3
1.1 Моделирование цифровых систем.	3
1.1.1 Общая характеристика цифровых систем.....	3
1.1.2 Проектирование ЦС.....	3
1.1.3 Моделирование ЦС.....	3
1.1.4 Типы (домены) моделей ЦС.....	4
1.1.5 Требования к средствам описания ЦС.....	7
1.1.6 Традиционные методы описания ЦС.....	8
1.2 Технология проектирования ЦС с использованием ЯСФО.....	10
1.2.1 Общие сведения.....	10
1.2.2 Основные свойства ЯСФО.....	10
1.2.3 Этапы проектирования на ЯСФО.....	10
1.2.4 Программирование.....	11
1.2.5 Тестирование проекта (модель тестирования):.....	13
1.2.6 Управление инструментальной средой.....	14
1.2.7 Компиляция.....	15
1.2.8 Симуляция проекта.	15
1.2.9 Синтез.....	16
1.2.10 Инструментальные средства ЯСФО-проектирования.....	17
2 ЯЗЫК VERILOGHDL.....	18
2.1 Общие сведения.....	18
2.2 Структура модели на VerilogHDL.....	18
2.3 Описание модуля.....	19
2.3.1 Структура модуля.....	19
2.3.2 Описание интерфейса модуля.....	20
2.3.3 Объекты и типы данных.....	21
2.4 Подключение модуля.....	22
2.4.1 Использование параметров.....	23
2.5 Построение вентильных моделей (Gate-level modeling) и примитивы.....	25
1.1.1 Предопределенные примитивы Verilog.....	25
1.1.2 Примитивы, определяемые пользователем (UDP).	26
1.1.3 Подпрограммы в Verilog: задачи (TASK) и функции (FUNCTION).	27
2.6 Системные задачи (system task).....	29
2.6.1 Системная задача \$display отображения данных и надписей.....	29
2.6.2 Системная задача \$monitor мониторинга состояния сигналов.....	30
2.6.3 Системные задачи управления симуляцией.....	31

2.7 Директивы компилятора.....	31
3 ЯЗЫК VHDL.....	32
3.1 Общие сведения.....	32
3.2 Структура VHDL-модели.....	32
3.2.1 Декларация объекта проектирования и описание интерфейса с внешним окружением....	32
3.2.2 Описание внутренней архитектуры объекта.....	33
3.3 Объекты и типы в VHDL.....	34
3.3.1 Объекты в VHDL.....	34
3.3.2 Типы в VHDL.....	35
3.3.3 Скалярные типы.....	35
3.3.4 Субтипы.....	37
3.3.5 Преобразование типов.	37
3.3.6 Составные (композиционные) типы.....	37
3.3.7 Тип access (доступ).....	40
3.3.8 Атрибуты.....	41
3.4 Выражения, операторы и симуляция VHDL-модели.....	42
3.4.1 Общие правила симуляции.....	43
3.4.2 Структура операторной части объекта.....	43
3.4.3 Поточковые выражения и задержки.....	44
3.4.4 Процессы (Process).....	47
3.4.5 Последовательные операторы.....	47
3.4.6 Параллельные выражения назначения сигналов.....	50
3.4.7 Параллельные выражения утверждения.....	50
3.4.8 Структурное описание цифровых систем. Подключение компонент (component installation).....	51

1 Общие принципы структурно-функционального описания аппаратуры.

1.1 Моделирование цифровых систем.

1.1.1 Общая характеристика цифровых систем.

Могут существовать различные взгляды на то, что является цифровой системой (ЦС). В рамках данного курса будем ориентироваться на следующее определение. *Цифровая система – любая дискретная электронная схема, обрабатывающая и запоминающая информацию.* При этом в качестве ЦС может рассматриваться и сложный комплекс, состоящий из нескольких блоков (например, ЭВМ), и каждый блок в отдельности.

С технической точки зрения современные цифровые системы характеризуются:

1. Блочной-иерархической организацией: системы состоят из подсистем и блоков, выделенных, в основном, по функциональному признаку. Блоки организованы в иерархию – крупные блоки состоят из более мелких блоков, те в свою очередь из еще более мелких и т.д. Нижний уровень иерархии – неделимые блоки, элементы.
2. Высокой структурной сложностью: значительным количеством структурных блоков/элементов и блочно-иерархических уровней. Уровней можно явно выделить до десятка и иногда больше. Блоков/элементов до тысяч – сотен миллионов. Рост сложности во времени продолжается в геометрической прогрессии. С целью облегчения понимания и проектирования столь сложных структур стараются обеспечить многократное использование одинаковых блоков в системе (например, используют ограниченный набор логических микросхем для реализации разных функций в схеме), а также стараются унифицировать интерфейс различных типов блоков.
3. Ориентацией на интегральное исполнение вплоть до завершенных вычислительных систем выполненных в виде одной микросхемы.
4. Высоким быстродействием – единицы и доли наносекунд.

1.1.2 Проектирование ЦС.

В процессе проектирования современных ЦС обычно решается классическая для любого проектирования задача соблюдения баланса трех показателей: технических параметров (структурно функциональных и динамических), цены, скорости проектирования/производства.

Достижение приемлемых результатов по трем направлениям в комплексе возможно только *при условии жесткой формализации и максимальной автоматизации процесса проектирования.* Для этого используются специальные методы и методики. В качестве базовой методики проектирования ЦС рассматривается их моделирование.

1.1.3 Моделирование ЦС.

Модель любой системы – это ее формальное описание, то есть в рамках фиксированной нотации (языковой, математической, графической или другой). Модели создаются для различных целей – описания, симуляции, верификации и других.

В процессе разработки ЦС путем моделирования решаются следующие задачи:

1. Фиксация требований к ЦС. Формальное представление позволяет с помощью специальных методов и инструментов проанализировать, обнаружить и исключить неполноту и двусмысленность описания проектных требований.
2. Тестирование и верификация ЦС, используя синтаксический анализ описания, контроль функционирования по математическому описанию или с помощью симуляции (имитационное моделирование). Дешево, легко и быстро проверяется соответствие поведения ЦС зафиксированным требованиям без выпуска пробного образца.

3. Автоматический синтез реализации ЦС в заданном базисе: в виде электронной схемы из типовых микросхем, в виде схемы из типовых логических ячеек или логических элементов определенного типа ПЛИС и базовых матричных кристаллов.
4. Получение формального описания функциональности и внутренней структуры системы. Такая модель используется для технического документирования ЦС.

1.1.4 Типы (домены) моделей ЦС.

Систему можно рассматривать с различных точек зрения, акцентируя внимание на том или ином свойстве. В соответствии с этим бывают модели различных типов.

Для ЦС обычно выделяют следующие типы моделей:

1. Функциональные – система описывается алгоритмами ее поведения, обеспечивающими реализацию заданных функций. При этом функциональность описывается в явном виде в форме «программы».
2. Структурные – система описывается в виде совокупности взаимосвязанных подсистем-блоков. Функциональность ЦС вытекает из совокупной функциональности составляющих подсистем и блоков. При этом функциональность блоков может не описываться в явном виде в проекте ЦС, но может иметь сторонние описания (например, в виде документации на используемые микросхемы).
3. Геометрические – описывается физическое расположение и геометрические параметры отдельных подсистем, например, на подложке микросхемы.

Вне зависимости от типа модели могут рассматривать систему с большей или меньшей степенью детализации: от уровня крупных функциональных блоков до уровня дискретных электронных элементов (для ЦС элементами будут транзисторы, резисторы и т.д.). В соответствии этому кроме деления по типам выделяют деление моделей по уровням абстракции (детализации). Модели одного типа, но различных уровней абстракции образуют домен, обозначаемый по типу моделей. Функциональный домен объединяет функциональные модели различных уровней абстракции, структурный – структурные, геометрический – геометрические.

Использование моделей различного типа определяется задачей проектирования: функциональное проектирование, конструкторское проектирование, схемотехническое проектирование и др. Уровень абстракции определяется этапом в едином потоке проектирования, например, общее архитектурное проектирование, разработка электронных модулей или «прошивок» для ПЛИС, тестирование ЦС и т.п.

Ниже представлена таблица, описывающая, как, из каких базисных элементов построены модели ЦС различных типов и уровней абстракции.

Уровень абстракции	Описательный базис модельных доменов		
	Функциональные модели ЦС	Структурные модели ЦС	Геометрические модели ЦС
системный (system)	высокоуровневые функции системы, без деталей их реализации.	абстрактные функциональные модули ЦС, связи между ними.	-
модульный (module)	высокоуровневые функции системы, без деталей их реализации, сгруппированные по модулям.	Модули вычислителей, памяти, ввода-вывода, коммутации	Зоны размещения модулей ЦС.
регистровых	переменные,	регистры,	стандартные ячейки

пересылка (RTL)	арифметические и логические выражения, управляющие конструкции	мультиплексоры, АЛУ и т.п..	
вентилей (gate)	булевы выражения, таблицы истинности	логические элементы	стандартные логические элементы, реальные или виртуальные
схемный (circuit)	дифференциальные уравнения	транзисторы, резисторы, и т.п.	Полигоны

1.1.4.1 Системный уровень.

На системном уровне функциональная модель представляет собой описание общего алгоритма функционирования системы. Обычно он представляется в форме программы (допускается и неформальное описание), использующей высокоуровневые функции цифровой системы, например: «считать значение из канала ввода», «отобразить данные», «сохранить данные» и т.п. Способ и средства реализации данных функций не описываются: преобразование данных не описывается математическими выражениями, не описывается сохранения значений в конкретных адресах памяти, нет привязки к типам данных или операциям конкретного процессора. Иначе говоря, функциональная модель на системном уровне описывает поведение системы, но не описывает способа реализации этой системы в виде аппаратуры или программного обеспечения. В связи с этим, такие модели называют «поведенческими» (behavioral).

Структурные модели системного уровня фиксируют подсистемы и связи, определенные поведенческой моделью. Элементами структурной системной модели (функциональными подсистемами) могут являться, например, система ввода температуры или других параметров, преобразователи данных различных типов, система хранения данных. Связи могут быть управляющими или по данным. Физическая структура ЦС и физические связи никак не отражаются в данной модели.

Геометрическая модель рассматривает геометрические размеры системы и ее элементов. Обычно на системном уровне геометрия системы не рассматривается, так как системный уровень не предполагает никакой связи с физической реализацией, в то время как геометрическая модель имеет неразрывную связь с физическим пространством.

Следует заметить, что при реальном проектировании ЦС конструктивные параметры системы (габариты, расположение блоков и др.) все равно приходится учитывать на самых ранних этапах, опирающихся на системные модели. На настоящий момент деятельность ведущих фирм в области систем проектирования направлена на создание единого цикла проектирования, что неизбежно приведет к «официальному» появлению конструктивных (геометрических) моделей на системном уровне.

Из практики проектирования к системному уровню могут быть отнесены функциональные схемы.

1.1.4.2 Модульный уровень.

Модели модульного уровня являются более детализированным вариантом системных моделей. На этом уровне появляется понятие физического модуля с закрепленными за ним функциями, занимающего определенное место в конструкции системы. Однако, как и на системном уровне, детали внутреннего устройства модуля не рассматриваются и не фиксируются. В связи с тем, что на практике разработка системного и модульного

описания часто выполняются параллельно и в жесткой связи, в некоторых источниках эти уровни абстракции объединяются в один – системный.

Функциональные модели модульного уровня схожи с функциональными моделями системного уровня, но дополнительно выполняется разделение функций на группы в соответствии с тем, как их предполагается распределять по модулям будущей ЦС в дальнейшем. Например, функции ввода-вывода закрепляются за процессором ввода вывода, функции хранения данных и контроля ошибок закрепляются за блоком памяти и т.д.

Структурные описания микропроцессорных систем на модульном уровне обычно представляют собой соединение вычислителя (процессора), памяти, модулей ввода-вывода, схем коммутации между этими блоками. В связи с этим такие модели называют Processor-Memory-Switch-модели (PMS-модели). Для PMS-модели показывают связи между перечисленными блоками. Для блоков явно или неявно (то есть в сторонней документации) фиксируется набор высокоуровневых функций (см. пример выше), однако не конкретизируют тип блоков или микросхем (если только на уровне справочной информации), не описывают физические сигналы электронной схемы.

Геометрические модели модульного уровня описывают размещение блоков, описанных PMS-моделью, в корпусе устройства, на плате или на подложке микросхемы. Примером таких моделей, особенно для микросхем, может быть «план размещения (floor plan)», на котором показаны места расположения процессорного ядра, памяти, контроллеров ввода-вывода. Floor plan может выдаваться большинством САПР спецмикросхем (ASIC) и ПЛИС. Другой пример – карта зон размещения процессорного ядра, схем памяти и ввода вывода на печатной плате. Цель создания таких обобщенных геометрических моделей – оценка площади платы или кристалла, оценка и оптимизация числа и протяженности связей между блоками, оценка тепловых режимов работы схемы, электромагнитного влияния соседних блоков и др.

1.1.4.3 Уровень регистровых пересылок.

На третьем уровне абстракции – уровне регистровых пересылок (Register Transfer Level (RTL) - описание ЦС выполняется в терминах элементов хранения и преобразования данных. В качестве первых рассматриваются регистры или их математический аналог переменные, в качестве вторых выступают функции или функциональные преобразователи, их выполняющие. На данном уровне выделяют потоки данных (data flow), передаваемые по определенным путям (data path), и секции управления этими потоками-путями (control section). Задача RTL – формальное, однозначное описание функциональности и внутренней структуры ЦС. RTL-описания пригодны для генерации (в том числе автоматической) электрической схемы или «прошивки» ПЛИС.

Функциональная RTL-модель описывает потоки данных в терминах арифметических или логических выражений и операций присваивания, а секции управления в виде управляющих структур или неявно, например, в форме правил последовательности выполнения операций в том или ином языке. Такую модель также называют потоковой.

Структурная модель представляет регистры, функциональные преобразователи (например, АЛУ), блок управления, а также линии управления и пути передачи данных между этими элементами.

Геометрическая модель показывает размещение и связи микросхем на печатной плате или стандартных логических ячеек (Logic Cell) на подложке микросхемы.

1.1.4.4 Уровень вентиляей.

На четвертом уровне абстракции – уровне вентиляей (gate), который также называют логическим уровнем (logical level), ЦС описывается в терминах булевых операций и битовых данных. В соответствии с этим функциональная модель представляет собой

набор булевых функций и таблиц истинности. Структурная модель – электрическую логическую схему в логическом базисе, например, И-НЕ, ИЛИ-НЕ или другом. Геометрическая модель представляет расположение и связи логических элементов, явных (как, например, в ПЛМ) или виртуальных, как, например, в ПЛИС типа FPGA, построенных на базе таблиц преобразования (Look Up Table (LUT)).

Вентильные описания по назначению аналогичны RTL-моделям, но более детально, фактически однозначно определяют электрическую схему ЦС. Такие описания необходимы в определенных технологиях проектирования ЦС, например, при использовании ПЛМ, при разработке специализированных микросхем (ASIC). В дополнение к этому, функциональная модель GATE-уровня является математическим описанием и пригодна для операций математической обработки: различных преобразований, минимизации.

1.1.4.5 Схемный уровень.

Наконец, на схемном уровне (Circuit level) описание ЦС выполняется в виде элементарных электронных элементов – транзисторов, диодов, резисторов. Их функционирование (то есть функциональная модель) представляется системой дифференциальных уравнений. Структура описывается электрической принципиальной схемой в терминах этих элементов. Геометрическая модель представляется в виде полигонов на подложке ИМС.

Еще раз отметим, что тип используемой модели определяется решаемой задачей (разработка принципиальной схемы, разработка конструкции, расчет тепловых режимов) и используемым в разработке базисом (платформой) (ПЛИС, набор логических ИМС, модели в библиотеке САПР).

В свою очередь уровень абстракции модели определяется выполняемым этапом проектирования:

- Системные и модульные описания используются на этапах общей архитектурной проработки;
- RTL- и GATE-модели являются базовыми при создании принципиальной электрической схемы, проектов для ПЛИС или полузаказных микросхем.
- Модели схемного уровня используются при подготовке производства микросхем.

В большинстве случаев разрабатывается совместно или последовательно несколько моделей. При этом переход от одной модели к другой является не всегда однозначным как по причине структуры (совместимости) этих моделей, так и по причине необходимости изменений при подобном переходе.

1.1.5 Требования к средствам описания ЦС.

Для разработки эффективных, как с точки зрения широты охвата свойств системы, так и с точки зрения качества, скорости и удобства проектирования, и достоверных моделей ЦС различных типов и уровней необходимо использовать специальные средства описания ЦС – графические, текстовые, математические и другие языки. Базируясь на анализе свойств современных ЦС, целей проектирования и моделирования, можно определить требования к современным средствам описания:

1. Жесткая формализация, стандартизация языков описания ЦС.
2. Поддержка многоуровневых описаний: едиными языковыми средствами должны создаваться модели различных уровней абстракции, от максимально высокого, желательно системного, до приемлемого уровня реализации, например, до уровня регистровых пересылок или логических элементов.
3. Поддержка описаний различных типов (доменов).
4. Единообразие и совместимость моделей различных типов и уровней.

5. Возможность несложного взаимного преобразования моделей различных типов и уровней. Поддержка поэтапного проектирования с поочередной детализацией (снижением уровня абстракции) моделей блоков и подсистем.
6. Семантическая и синтаксическая совместимость с распространенными языками программирования.
7. Ориентация на автоматизированную обработку и минимизацию «ручных» операций.
8. Описание больших по объему ЦС: краткость, легкость восприятия, иерархичность.
9. Возможность статической и динамической верификации проектов.
10. Возможность распараллеливания работ при проектировании (при разработке, анализе и тестировании моделей).
11. Средства организации проекта (структуризация, документирование).
12. Средства хранения и повторного использования наработок.
13. Переносимость проектов между различными инструментальными системами.

1.1.6 Традиционные методы описания ЦС.

К традиционным методам описания можно отнести:

1. графические структурные и принципиальные схемы;
2. текстовые структурные описания;
3. булевы выражения и преобразуемые в них таблицы истинности, таблицы переходов.
4. языки структурно-функционального описания.

Графические структурные и принципиальные схемы

Основные достоинства:

- легкость восприятия;
- совместимые описания структур на различных уровнях абстракции: от системного до схемного;
- поддержка иерархии, позволяющей совмещать в одном проекте описания различных уровней, проектировать блоки поэтапно.

Основные недостатки:

- Низкая степень формализации, затруднен перенос между различными инструментальными средствами;
- Только структурные описания.
- Ограниченная сложность схем (по некоторым оценкам – до 6000 элементов);
- Сложность автоматической обработки.

Текстовые структурные описания.

Это текстовая интерпретация структурных и принципиальных схем. Используется, в основном как средство переноса проектов между инструментальными пакетами. Наиболее известен стандарт EDIF (Electronic Design Interchange Format). Текстовые структурные описания также можно выполнять на языках структурно-функционального описания, таких как, VHDL, VerilogHDL.

Относительно схемных описаний, к достоинствам относится:

- жесткая формализация и стандартизация описаний, простота переноса;

К недостаткам добавляется:

- сложность восприятия.

Булевы выражения.

Почти любые цифровые схемы включают блоки комбинационной логики. Например, адресный селектор – чисто комбинационная, схема цифрового автомата – последовательная включающая кроме комбинационных блоков элементы памяти. Для разработки комбинационных схем используются их описания в виде булевых выражений.

Такие формы описаний, как таблицы истинности и таблицы переходов-выходов однозначно и формальными методами преобразуются в булевы выражения и относятся к этому же классу описаний.

Основные достоинства:

- формализованные описания;
- позволяют оптимизировать реализация путем аналитической обработки – минимизации булевых выражений.

Основные недостатки:

- поддерживает только функциональное описание на уровне вентилей;
- требуется булево выражение для каждой битового результата: выхода схемы или входа триггера. Это делает описания больших схем очень громоздкими, трудно понимаемыми;
- не имеет специальных средств организации проекта.

Языки структурно-функционального описания.

На настоящий момент являются наиболее мощной и гибкой технологией разработки и описания цифровых систем. Позволяют описывать как алгоритм функционирования ЦС, так и ее структуру. Имеют синтаксис и семантику во многом родственную языкам программирования. Могут преобразовываться в графические изображения системы (схемы) и в булевы описания.

1.2 Технология проектирования ЦС с использованием ЯСФО.

1.2.1 Общие сведения.

Языки структурно-функционального описания ЦС (ЯСФО) появились в 60-х годах, как средство формализованного документирования (описания) структуры и поведения ЦС. Язык VHDL (Very high speed integrated circuit HDL - язык описания аппаратуры высокоскоростных интегральных схем), разработанный МО США на базе языка ADA, был первым стандартом в этой области ANSI/IEEE Std 1076-1987, ANSI/IEEE Std 1076-1993, ANSI/IEEE Std 1076a-2000). VHDL является жестко типизированным языком, имеет достаточно сложный, «многословный» синтаксис.

Еще одним стандартом в этой области является язык VerilogHDL, стандартизованный IEEE-1364-1995/2001. VerilogHDL разрабатывался для проектов схем на ПЛИС, где не требуется высокоуровневых абстрактных описаний системы, но язык должен быть кратким, выразительным, многофункциональным. Он имеет простой C-подобный синтаксис. В настоящее время на Verilog-е проектируют многие крупнейшие фирмы-разработчики микросхем, например, Motorola.

Кроме того, некоторые фирмы – производители микросхем программируемой логики (сейчас это – одна из ведущих прикладных областей HDL) разработали свои языки, ориентированные на специфику организации их ПЛИС, например, AHDL фирмы Altera, AbelHDL фирмы Xilinx. Однако к настоящему времени значимость, поддержка и развитие этих нотаций практически сошли на нет.

1.2.2 Основные свойства ЯСФО.

ЯСФО были специально разработаны и удовлетворяют описанным выше требованиям в современным средствам описания ЦС.

Язык VHDL позволяет описывать цифровые схемы вниз до уровня ячеек (GATE). Наряду с этим VHDL имеет средства создания и работы с данными произвольных типов, в том числе физических. Например, это может быть бит, битовый вектор, скалярные численные типы, время, длина, скорость, тип, перечисляющий состояния системы и т.д. За счет этого VHDL позволяет делать высокоуровневые абстрактные описания, то есть покрывает системный (SYSTEM) уровень.

Язык Verilog в значительном объеме покрывает уровни от модульного (MODULE) до уровня ячеек (GATE). Частично закрыт схемный (CIRCUIT) уровень – на нем можно описать транзисторный каскад, реализующий логические функции.

Большинство языков ЯСФО, в том числе VHDL и Verilog, покрывают только структурный и функциональный домены моделей. Описание и проектирование топологии выполняется с помощью иных, на нынешний момент не стандартизированных средств.

1.2.3 Этапы проектирования на ЯСФО

Создание и использование ЯСФО-модели является многоэтапным, итерационным процессом, который может включать в разном сочетании перечисленные ниже основные шаги.

1. Программирование (programming) - разработка модели (программы) на ЯСФО.
2. Компиляция (Compilation) – преобразование исходной модели в исполняемый (симулируемый или синтезируемый) промежуточный код. Промежуточный код сохраняется в проектных библиотеках.
 - 2.1. Анализ (Analysis) - синтаксический и семантический контроль исходных текстов, интерпретация директив компиляции.
 - 2.2. Уточнение, разбор (elaboration) – преобразование модели ЦС в иерархию поведенческих описаний – процессов, которые могут быть симулированы.

Преобразование выполняется путем подстановки вместо «вызовов» объектов их внутренних описаний по иерархии сверху вниз, а также заменой (макроподстановкой) компонентов структурных описаний их поведенческими аналогами (например, вместо логического элемента «ИЛИ» подставляют конструкцию IF-THEN-ELSEIF).

3. Симуляция (Simulation) или Исполнение (Execution) – исполнение промежуточного кода, полученного после компиляции. Симуляция может выполняться в несколько этапов:
 - 3.1. Функциональная симуляция, без учета временных задержек;
 - 3.2. Временная (полная) симуляция цифровой схемы.
4. Синтез (Synthesis) – преобразование исходной модели в некоторый заданный базис элементов (например, базис логических элементов), которые могут быть реализованы «исполняющими» средствами, например, микросхемой ПЛИС.
5. Реализация (implementation) – отображение синтезированной модели на элементы и связи конкретной электрической схемы – на электронную плату, ПЛИС или заказную микросхему. В качестве подэтапов могут появляться подгонка (FITTING), программирование (PROGRAMMING) и др.

Состав и порядок выполнения перечисленных этапов в конкретной работе может изменяться в зависимости от решаемой задачи и выбранной технологии проектирования и составляют *технологическую структуру проекта*. Варианты организации этих этапов для наиболее распространенных типов проектов - симуляции ЦС и для проектирования на ПЛИС - приведены ниже **на рисунке**.

1.2.4 Программирование.

На этапе программирования создается модель цифровой системы и организуется проектное окружение. Модель ЦС разрабатывается исходя из целей выполнения проекта – описание структуры ЦС, описание или верификация функциональности, разработка структурного описания, используемого как головное описание в рамках схемотехнического проекта, ПЛИС-реализация. На этих же основаниях выбирается технология проектирования, включая состав и последовательность этапов, инструментальные средства.

Создание модели цифровой системы включает разработку архитектуры ЦС, разработку тестового окружения, кодирование на одном из языков (например, VerilogHDL или VHDL).

Проектирование архитектуры ЦС является сложным и слабо формализованным процессом. Использование ЯСФО-технологий не меняет его принципиально, но, исходя из особенностей представления ЦС в ЯСФО, рекомендуется применять определенные технологические подходы и технические решения, которые позволят достичь более качественного результата. К примеру, существуют обширные рекомендации по созданию асинхронных систем или по созданию синтезируемых проектов.

Создание проектного окружения подразумевает конфигурирование инструментальной среды, выбор и подключение внешних библиотек, создание командных скриптов, тестовых векторов для симуляции и т.д. Проектное окружение обычно включает платформенно-независимые (переносимые) компоненты, которые могут быть использованы в любой инструментальной среде проектирования, и платформенно-зависимые (непереносимые) компоненты, которые жестко привязаны к конкретной инструментальной среде. Компоненты первой группы обеспечиваются средствами основного языка описания (Verilog, VHDL) или вспомогательного (инструментального) языка управления инструментальной средой (например, TCL/TK) или переноса проектов

(например, EDIF). Компоненты второй группы определяются возможностями конкретной среды проектирования (например, файлы описания проектов, файлы описания графических блок-схем и временных диаграмм)

Нельзя сказать, что на этапе программирования создается завершенная модель ЦС, контролируемая и применяемая на последующих этапах проектирования (симуляция, синтез и т.д.). На самом деле, процесс создания модели ЦС является комплексным (т.е. охватывает различные аспекты и этапы проектирования), итерационным, а этап программирования выделяется только в качестве объединяющего и координирующего в этом сложном процессе.

Ниже представлен типовой поток проектирования ЦС, характерный для ЯСФО. В данном случае рассматриваются не технологические этапы, как было представлено выше, а этапы развития модели ЦС от постановки задачи до реализации.

1. Разработка входных и выходных тестовых векторов. Входные тестовые вектора представляют собой воздействия, подаваемые на входные порты моделируемой системы. Важно, чтобы входные вектора описывали различные, лучше все возможные, ситуации старта ЦС, в том числе учитывая варианты внутренних состояний системы. Например, должны отрабатываться ситуации рестарта системы в процессе работы, когда во внутренние регистры занесены некоторые значения. Выходные вектора представляют образец ожидаемого поведения системы при воздействии на нее входных векторов. В процессе отладки системы нужно будет сравнивать симулируемое поведение ЦС с выходными тестовыми векторами и, исходя из результатов, судить о корректности функционирования ЦС. Таким образом, выходные вектора не подаются на входные порты ЦС и жестко связаны с входными векторами. Входные и выходные вектора формируются из исходных требований к проектируемой ЦС. Иными словами, это задание на проектирование, формализованное в терминах ЯСФО. Вместе множество входных и выходных тестовых векторов принято называть термином **«golden vectors»**. Так как не возможно сразу полностью и безошибочно представить и описать поведение сложной системы, то тестовые вектора будут корректироваться на всем протяжении разработки системы.
2. Разработка поведенческой модели системы и взаимная отладка совместно с golden vectors. На данном этапе разрабатывается поведенческое описание ЦС. Оно состоит из одного или нескольких модулей (блоков), но, обычно, не имеет иерархической структуры. Такое описание проверяется (верифицируется) с помощью golden vectors. Цель данного этапа – укрупнено определить структуру ЦС и функции ее подсистем.
3. Декомпозиция системы на более простые блоки. На этом этапе строится иерархическая модель ЦС. Сложные блоки разбиваются на более простые субблоки, для которых разрабатываются поведенческие или структурные описания. В дальнейшем данный процесс может повторяться для субблоков.
4. Разработка субблоков. На данном этапе выделенные субблоки разрабатываются независимо друг от друга одним или несколькими разработчиками. Могут быть предложены несколько реализаций одного субблока, как поведенческие так и структурные. Поведенческие обычно разрабатываются за более короткие сроки и быстрее работают в процессе симуляции. Поэтому они используются для совместной симуляции с другими субблоками в рамках полной «сборки» системы. Структурные модели представляют собой синтезируемую реализацию системы, однако более трудоемки в разработке и отладке. Они обычно создаются после и на основе отлаженной поведенческой модели.

5. Совместное тестирование submodule системы на основе golden vectors. Этот этап завершает итерационную петлю из предыдущих двух. Выполняется верификация совместной работы всех блоков и subблоков ЦС на основе golden vectors.

1.2.5 Тестирование проекта (модель тестирования):

Задачу проверки корректности проекта можно разделить на два этапа:

1. Статическая проверка – синтаксический контроль текста и семантический контроль структуры модели, осуществляемые на этапе анализа и компиляции проекта. Данный этап является обязательным – без него невозможно выполнение симуляции - и выполняется автоматически инструментальными средствами (компилятором ЯСФО).
2. Динамическая верификация – контроль правильной реакции ЦС на различные наборы и последовательности входных сигналов. Такие тестовые входные воздействия – стимуляторы (stimuli) – должны быть поданы на все входы ЦС в определенной комбинации, а комбинации должны сменяться во времени в определенном порядке и с определенными задержками. В простых случаях порядок тестовых комбинаций фиксирован, они подаются друг за другом в определенной последовательности. Таким образом, можно протестировать, например, комбинационную схему. При тестировании более сложных систем, например, цифровых автоматов, порядок смены тестовых воздействий может изменяться в зависимости от реакции ЦС на предыдущие воздействия, что должно отражать реакцию внешней среды и устройств на поведение цифровой системы.

Для решения задачи динамической верификации модели ЦС, ЯСФО поддерживают механизм Test Bench («Испытательный стенд»). Test Bench – это дополнительное внешнее окружение ЯСФО-модели ЦС, которое позволяет формировать тестовые входные воздействия, подавать их на входы тестируемой системы, анализировать корректность реакции ЦС на тестовые воздействия, отображать и сохранять результаты работы ЦС – состояния выходных и внутренних сигналов ЦС – и результаты анализа работы ЦС в виде сообщений и кодов ошибок.

- Test Bench оформляется в виде модуля на ЯСФО, а не является дополнительной инструментальной системой.

Механизм Test Bench оформляется как независимый модуль проекта (module – в Verilog или entity/architecture – в VHDL) и состоит из следующих элементов:

- генератор тестовых воздействий – подсистема, самостоятельно формирующая тестовые воздействия или считывающая их из внешнего файла;
- сокет для подключения тестируемой цифровой системы – Unit under test (UUT);
- монитор реакции UUT с функциями отображения, сохранения состояний и анализа выходных сигналов.

Генератор тестовых воздействий – один или несколько поведенческих блоков (initial, always (Verilog) или process (VHDL)), в которых описывается последовательность изменения входных сигналов. Большинство современных инструментальных средств позволяет автоматически создавать блоки генераторов из графических временных диаграмм или из файлов с текстовым описанием изменения входных сигналов во времени. Файлы с текстовым описанием входных воздействий также можно подключать непосредственно к проекту Test Bench без преобразования в текст генератора на ЯСФО.

Во многих случаях используют также генераторы эталонной реакции. Они предназначены для формирования образца требуемой реакции UUT и последующего автоматического сравнения с результатами тестирования.

Сокет для подключения UUT – интерфейс между генератором воздействий и UUT. Представляет собой ЯСФО-выражение подключения структурного элемента (инсталляции компонента). В качестве подключаемого компонента указывается UUT, а в качестве внешних интерфейсных сигналов – выходы генератора воздействий.

Монитор реакции – функции монитора поддерживаются частично внутренними средствами ЯСФО, а частично – внешними инструментальными средствами (симулятор, анализатор временных диаграмм и др.). Часть, встроенная в ЯСФО-проект – это набор команд управляющих отображением и сохранением в log-файле состояний выходных сигналов (команды \$display, \$monitor, \$fmonitor (Verilog) или Assert/Report (VHDL)), а также выполняющих простой анализ результатов путем обнаружения недопустимых состояний тестируемой ЦС или сравнения с эталоном реакции (Assert/Report (VHDL)). Внешние инструментальные средства позволяют гибко управлять процессом симуляции-тестирования в интерактивном режиме или в пакетном режиме, под управлением командных файлов не на ЯСФО.

1.2.6 Управление инструментальной средой.

Большинство распространенных ЯСФО имеют два типа средств взаимодействия с инструментальной средой (с программным пакетом, в котором выполняется разработка и симуляция модели):

1. Директивы компилятора, включая текстовый препроцессор;
2. Системные функции.

Директивы компилятора включают функции настройки параметров компиляции, например, выбор единиц измерения по умолчанию (функция 'timescale (Verilog)), выбор типа цепей по умолчанию (функция 'default_nettype (Verilog)), а также функции текстового препроцессора, аналогичные препроцессорам языков программирования:

- условная компиляция ('define, 'ifdef (Verilog) или if-generate (VHDL);
- выполнение макроподстановок;
- включения (include) текста;
- подключение библиотек.

Системные функции обеспечивают сбор/анализ информации и управление инструментальной системой в процессе симуляции. Типичный набор системных функций поддерживает:

- работу с файлами (запись/чтение данных в файл, сохранение данных трассировки в файле и др.);
- вывод строк, параметров, сообщений;
- отслеживание событий при симуляции (изменение значения переменной или состояния сигнала, определенное значение переменной или сигнала, недопустимое (ошибочное) значение);
- управление процессом симуляции (старт, стоп);
- расчет математических функций.

Кроме стандартных функций можно определять специфические пользовательские функции. Такая возможность поддержана в языке VerilogHDL с помощью стандартного интерфейса PLI (Program Language Interface), который позволяет подключать к симулятору функции сторонней разработки, например, математические функции, написанные на языке С.

Функции управления инструментальной средой различными ЯСФО поддерживаны в большей или меньшей степени и имеют различную реализацию. Они наиболее развиты в VerilogHDL и выделены в отдельные классы со специфическим синтаксисом (префикс ' указывает на функции управления компилятором, а префикс \$ - на системные функции).

Язык VHDL также обеспечивает большинство механизмов управления инструментальной средой, однако, здесь они менее развитые и четкие. Не всегда есть четкое выделение функций данных типов. Они оформляются по общим правилам, что снижает читаемость текста (сложнее различать «описание модели» и «управление инструментальной системой»). Кроме того, разнообразие функций меньше и их назначение обозначено менее четко (например, `if – generate` для задач условной компиляции или `assert-report` для задач мониторинга).

1.2.7 Компиляция.

1.2.8 Симуляция проекта.

Исполнение или симуляция - ключевой этап при создании проектов на ЯСФО. Можно рассматривать ее как эквивалент исполнения программного проекта. Понимание механизма симуляции модели (программы) на ЯСФО необходимо для корректного использования операторов и выражений ЯСФО, для построения более простых и элегантных программ, для устранения многих ошибок, возникающих при симуляции и не всегда ясных и объяснимых с точки зрения правил цифровой схемотехники.

По технологии исполнения симуляция может быть программная – на кросс-компьютере, или аппаратная – средствами программируемой логики инструментальной системы. Последний вариант более затратный и используется в случае потребности в высокой скорости симуляции, недостижимой программно. Например, если необходимо смоделировать сложный процессор.

По учету инерционных свойств аппаратуры можно выделять функциональную симуляцию (*functional simulation*), не учитывающую задержек распространения и обработки сигналов и временную симуляцию (*timing simulation*), учитывающую задержки распространения и обработки сигналов. Функциональная симуляция удобна и применяется для отладки логики работы цифровой системы. При этом учитываются только временные задержки явно указанные в HDL-тексте. Например, в VHDL-строке указана задержка сигнала на 10ns: `a<=b after 10 ns;`. Временная симуляция используется при реализации проекта на определенных аппаратных средствах – на ПЛИС, на ASIC или микросхемах стандартной логики. Она учитывает задержки, вносимые логическими элементами и более сложными блоками – ячейками ПЛИС или ASIC, а также задержки распространения сигналов по линиям соединений элементов. В связи с этим результаты временной симуляции определяются типом и быстродействием блоков (ячеек) конкретной используемой ПЛИС или ASIC, длиной и структурой соединений (трассировкой) внутри ПЛИС или ASIC. Для описаний типов внутренних ячеек микросхем, их временных параметров, используются библиотеки, предоставляемые производителем микросхем. Библиотеки написаны на языке Verilog с использованием UDP (User Defined Primitive) или на базе специальной библиотеки VHDL – VITAL (VHDL Initiative Toward ASIC Libraries). Для передачи информации о задержках в конкретной трассировке от программ трассировки используются описания в формате SDF (Standard Delay Format).

По применяемым алгоритмам симуляции, бывает Time-based-симуляция (на базе непрерывного времени), Event-Based-симуляция (на базе дискретных событий), Cycle-Based- симуляция (на базе циклов функционирования).

Механизм Time-based симуляции подразумевает периодическое, с минимальным шагом обновление состояния симулируемой системы. Например, если для Verilog-проекта с помощью директивы `'timescale` минимальная единица времени определена в 1 ps, то в процессе симуляции состояние всех сигналов системы от входов до выходов будет просчитываться с шагом в 1 ps независимо от того, были ли изменения в их состоянии или нет. Такой метод симуляции необходим и применяется для аналоговых схем (так называемый метод SPICE), где сигналы непрерывно изменяются, но неэффективен –

медленный - для реальных цифровых схем, где изменение сигналов происходит относительно редко, по сравнению с максимально достижимой интенсивностью.

Механизм Event-based (событийной) симуляции обрабатывает только изменения состояния схемы, которые могут происходить либо в результате изменений внешних сигналов, либо в результате появления внутренних событий (работа внутренних генераторов тактовых импульсов). Такой механизм удобен для моделирования цифровых систем, так как не требует постоянного «обсчета» системы, а только в редкие моменты возникновения событий. Особенно удобен для автоматных систем, когда тактовый генератор определяет событие смены состояния системы.

Однако, учитывая высокие частоты внутренних генераторов, интенсивность event-based симуляции может оставаться достаточно высокой, так как обновление состояний происходит на каждом перепаде тактового сигнала. Например, если схема не автоматного типа, перепады не вызывают изменения состояний схемы, так как они просто синхронизируют входные и выходные сигналы. В таком случае нужно ориентироваться на cycle-based-симуляцию.

Механизм cycle-based-симуляции (циклической) подразумевает изменение внутреннего состояния схемы только в результате изменений внешних сигналов, а внутренние события (от генераторов, петель фазовой подстройки (PLL), часов реального времени) не обрабатываются. Для каждого состояния внешних сигналов вычисляется внутреннее состояние системы, запоминается и при следующем возникновении данной комбинации повторного пересчета не выполняется, а используются сохраненные результаты. Циклическая симуляция подразумевает синхронизацию изменений входных и выходных сигналов между собой, не обрабатывает задержек распространения сигнала, пригодна только для комбинационных схем, но зато очень быстро выполняется.

1.2.9 Синтез.

Синтез (synthesis) – преобразование исходной модели в некоторый заданный базис элементов (например, базис логических элементов), которые могут быть реализованы «исполняющими» средствами, например, микросхемой ПЛИС.

В процессе синтеза описания всех уровней и типов (доменов), присутствующие в исходной модели, преобразуются в единообразное описание определенного уровня и типа. Для задач проектирования электронных цифровых схем, в том числе ПЛИС и ASIC, обычно выполняется синтез структурных RTL-описаний или GATE-описаний, что соответствует внутренней структуре современных цифровых СБИС. Такой синтез часто называют структурным. Синтез идеологически и по структуре этапов схож с этапом компиляции (синтаксический и семантический контроль, преобразование базиса). Однако, во всяком случае на современном уровне развития структурного синтеза, не возможно выполнить однозначные преобразования любых произвольных поведенческих моделей в структурные элементы. Поэтому используются шаблонные макроподстановки. Например, за место конструкции IF-THEN-ELSEIF подставляют мультиплексор. Для сложных конструкций такая подстановка далеко не однозначна и может быть ошибочной. Именно в этих неоднозначностях, а также в эффективности макроподстановки с точки зрения быстродействия, объема электронной схемы и других и заключаются проблемы структурного синтеза.

При создании и реализации алгоритма синтеза следует учитывать, что у различных семейств ПЛИС и ASIC имеются различные типы и количества структурных элементов (логических элементов, триггеров, генераторов, регистров сдвига для построения контроллеров линий связи и т.п.), что эти элементы могут быть реконфигурируемыми (например, таблицы перекодировки (look up table, LUT). Хранить и эффективно применять для синтеза библиотеку макроподстановок для каждого типа ПЛИС или ASIC не

представляется возможным, так как этих типов может быть несколько тысяч. Поэтому используют двухшаговый механизм:

1. Сначала исходное HDL-описание преобразуют в унифицированный базис типовых структурных элементов – логических элементов, триггеров, мультиплексоров и т.д. На данном этапе применяют сложные алгоритмы структурного синтеза из поведенческих или смешанных описаний.
2. Элементы промежуточной структуры в унифицированном базисе отображают на структурные элементы конкретной микросхемы, применяя рекомендации или готовые алгоритмы/процедуры преобразования, поставляемые производителем микросхем.

Чтобы производители знали, под какой унифицированный базис готовить процедуры преобразования, этот базис закрепляют в специальных библиотеках и стандартах, например, в стандарте VITAL (для VHDL-синтеза).

Серьезной проблемой синтеза является необходимость учета задержек структурных элементов и соединительных цепей синтезированной системы. При этом в системе могут возникнуть события и состояния, не регламентированные исходной HDL-моделью. Например, в результате гонок в схеме, цифровой автомат в какой-то момент переключится в непредсказуемое состояние, что изменит работу схемы в целом. Синтезатор должен по возможности учитывать эти ситуации и строить структуры таким образом, чтобы их не возникало.

1.2.10 Инструментальные средства ЯСФО-проектирования.

В настоящее время выпускается и доступно широкому пользователю большое количество инструментальных систем различного назначения, часто комплексных:

- редакторы: HDLDesigner(MentorGraphics), ActiveHDL (Aldec) и др.
- компиляторы-симуляторы: ModelSim (MentorGraphics), ActiveHDL (Aldec), SynaptiCAD software (SynaptiCAD) и др.
- логические синтезаторы: LeonardoSpectrum (Exemplar/Mentor Graphics), FPGA Compiler (Synopsys), Synplify (Synplicity) и др.

Собственные сквозные средства проектирования, включающие редакторы, симуляторы и синтезаторы выпускают фирмы-производители ПЛИС: Quartus II, MAX+PLUS II (Altera), ISE, EDK (Xilinx).

Средства проектирования на VHDL, VerilogHDL, AHDL, AbelHDL входят в пакеты проектирования печатных плат и программируемой логики: Protel (Altium), PCB Design Studio/Allegro (Cadence) и другие.

Области применения ЯСФО:

- проектирование ПЛИС, ASIC, SOC.
- проектирование ЦС на печатных платах.
- тестирование ЦС.
- документирование ЦС.

На базе «классических» ЯСФО разработаны языки-расширения для смежных областей, например:

- для технологий граничного сканирования (загрузка программ и данных, тестирование)– BSDL;
- языки описания аналогово-цифровых схем, например, VHDL-AMS;
- объектно-ориентированное описание ЦС – ObjectiveVHDL, VHDL+.

2 Язык VerilogHDL.

2.1 Общие сведения.

VerilogHDL является (вместе с VHDL) одним из двух самых распространенных ЯСФО. Он разработан в 1983 году Филом Муром в подразделении ф. Cadence, с 1990 г. является основным языком в разработках Cadence и принимается основными фирмами в электронной промышленности, создающими организацию Open Verilog International (OVI). В 1995 г. VerilogHDL стандартизован IEEE-1364.

VerilogHDL – С-подобный язык, простой в понимании и изучении. По сравнению с VHDL характеризуется более компактными исходными текстами. Verilog ориентирован на синтезируемые модели для упаковки в ПЛИС и ASIC, а не на высокоуровневые описания, как VHDL, и в соответствии с этим покрывает нижние уровни абстракции: от системного до вентильного. Еще одно преимущество – развитая система управления инструментальными средствами, что также является преимуществом. Однако возможность расширения функций управления инструментальной средой наряду с расширением функций приводит и к появлению нескольких нотаций (OVI, VerilogXL (Cadence), ModelSim (Mentor Graphics), VeriWell (SynaptiCAD) и др.). Перечисленные достоинства привели к широкому признанию Verilog: в настоящее время по объему приложений Verilog начинает опережать VHDL.

2.2 Структура модели на VerilogHDL.

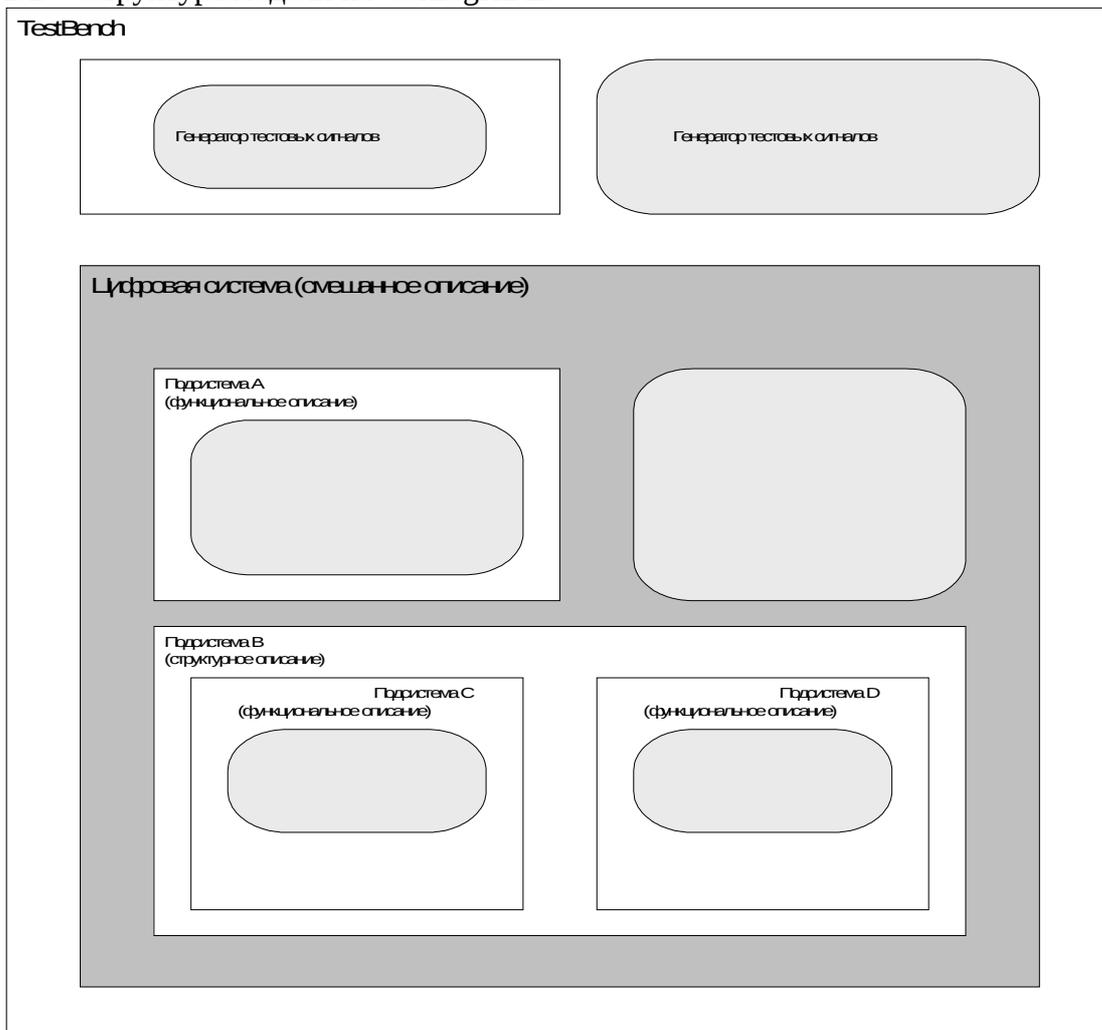


Рисунок 1

Модель (описание) цифровой системы на Verilog представляет собой многоуровневую иерархию компонент: подсистем, блоков и элементов, соединенных цифровыми сигналами, одиночными и шинами. Компоненты системы описываются в Verilog как модули (**module**). Модуль имеет явное описание функциональности и/или внутренней структуры. Также в Verilog используются примитивы (**primitive**) - компоненты, моделирующие функции цифровых электронных каскадов (логических элементов, ключей или каскадов, разработанных пользователем). Специфика примитивов в том, что, по сравнению с модулями, при синтезе электронных схем они рассматриваются как готовые элементы цифрового базиса.

Модули и примитивы соединяются между собой через внешний интерфейс, представляющий собой набор портов. Порты – это внешние сигналы модуля/примитива, одиночные или шины. В некоторых случаях, например, для модулей «testbench», модуль может не иметь портов.

Внутренняя структура и функционирование каждого модуля представлена функциональными (потокowymi и поведенческими) и структурными описаниями. В рамках одного модуля могут использоваться функциональные или структурные конструкции, а также их сочетание (смешанные описания).

Структурное описание внутреннего устройства модуля представляет собой множество более низкоуровневых модулей и сигналов, соединяющих порты моделей. В свою очередь каждый из этих модулей может быть описан таким же образом. Описанный один раз модуль может быть использован многократно на одном или разных уровнях модели системы, то есть создаются несколько образцов (**instances**) модуля данного типа.

Проект на Verilog (как и на других HDL) обычно организуется следующим образом (см. рисунок выше): на самом верхнем уровне иерархии существует только один «главный» модуль (подобно процедуре main в языке C). Обычно это модуль TestBench. Он, в свою очередь, включает один модуль - разрабатываемую ЦС и, возможно, модули генератора тестовых сигналов. На следующем уровне иерархии модуль ЦС распадается на любое количество модулей-подсистем и поведенческих описаний и так далее.

2.3 Описание модуля.

2.3.1 Структура модуля.

Модуль имеет следующую структуру:

<декларация модуля>	module <i>имя модуля (список портов)</i>
<декларация портов>	input, output, inout
<декларации параметров>	parameter
<декларация подключаемых файлов>	include
<декларации объектов>	wire, reg...
<параллельно исполняющиеся блоки, в т.ч.:	
параллельные выражения назначения (присваивания)	assign
процедурные (поведенческие) блоки	initial, always
вызовы задач и функций	task, function
подключение модулей и примитивов;	
>	
endmodule	

ПРИМЕР:

```
module REG4(q,qn,data,clk,clrn) //декларация модуля со списком портов

input [3:0] data; //декларация портов
output [3:0] q,qn;
input clk, clrn;

    DFF d0 (q[0],qn[0],data[0],clk,clrn); //подключение образцов модуля DFF
    DFF d1 (q[1],qn[1],data[1],clk,clrn);
    DFF d2 (q[2],qn[2],data[2],clk,clrn);
    DFF d3 (q[3],qn[3],data[3],clk,clrn);

endmodule
```

2.3.2 Описание интерфейса модуля.

Назначение интерфейса модуля - включение модуля как компонента в проект более сложной цифровой системы и соединение с другими модулями посредством сигналов, выходящих из модуля - портов.

Описание интерфейса модуля включает две части:

1. Список портов. После имени модуля в скобках перечисляются имена внешних портов модуля, разделенные запятой.
2. Декларация портов. Это первая часть декларативной части модуля. Здесь приводится подробное описание портов: их направление и размерность, если порт многоразрядный (вектор – в терминологии Verilog). Формат декларации порта:

направление_порта размерность_вектора имя_порта;

Бывают следующие направления портов:

input – входной; **output** – выходной; **inout** – двунаправленный.

Размерность указывается только, если порт является вектором.

Имя порта должно соответствовать имени в списке портов.

Если направление и размерность нескольких портов одинаковы, то они могут быть перечислены через запятую после общего описания.

Имеются определенные правила подключения портов к внутренним и внешним сигналам:

- 1) для портов **input** внешним сигналом может быть **net** (**wire**, **wand**, **wor**, **tri** и т.д.) или **reg**, а внутренним – только **net**;
- 2) для портов **output** внешний сигнал всегда **net**, а внутренний – **net** или **reg**. Чтобы **output**-порт стал типа **reg** – необходимо объявить сигнал типа **reg** с таким-же именем, как у порта.
- 3) для **inout** портов и внешний и внутренний сигналы – **net**.

В отличие от VHDL, порты не могут быть других (произвольных) типов, например, **integer** или **real**.

Пример описания интерфейса модуля:

```
module Processor (Clock, Reset, Address, Data, Read_Write)
input Clock, Reset;
output [19:0] Address;
inout [15:0] Data;
out Read_Write;
...
reg Read_Write;
...
endmodule
```

2.3.3 Объекты и типы данных.

Стандарт Verilog не разделяет объекты данных и типы данных. Под термином «Типы данных» понимаются (и описывается) как сами типы, так и связанные с ними объекты. Такой подход оправдан достаточно узкой специализацией Verilog на описание цифровых схем, когда количество различных пар «объект-тип» минимально и фиксировано (в отличие от этого, в VHDL-моделях системного уровня любой сигнал или переменная может иметь любой тип, в том числе - не реализуемый физически, в виде схемы. Например, значение физической величины напряжения, тока, или наименование состояний работы системы).

В Verilog типы данных разделены на группы, которые различаются назначением и правилами применения в моделях.

1. Цепи (сигналы) (**net**) - связывают компоненты – модули и примитивы. Цепи являются абстракцией физических соединений между модулями. Они имеют то значение, которое «на них подается» в данный момент времени и не могут запоминать значения после снятия внешнего воздействия. Исключения составляют цепи типа **triereg**, моделирующие цепи с емкостью. Цепи должны быть «запитаны» от других цепей, регистров или портов. Значение может присваиваться либо непрерывно (в выражениях **assignment** в теле модуля), либо на ограниченный промежуток времени (в выражениях **procedural assignment** в процедурных блоках). Последнее имитирует, например, питание цепи через отключаемый ключ.
2. Переменные (**variable**, по стандарту Verilog 2001) или регистры (**reg**) - элементы хранения данных, как переменные в языках программирования. Соответственно, регистрам нельзя присваивать значение «непрерывно» в потоковых выражениях. Значения регистрам присваивается ТОЛЬКО в процедурных блоках (**initial**, **always**).

Цепи делятся:

1. по мощности (по значению параметра **strength level**):
 - а) по мощности питающей цепи (drive strength):
 - supply
 - strong
 - pull
 - weak
 - highz
 - а) по мощности, потребляемой емкостной цепью при заряде (charge strength) (только для цепей типа **triereg**):
 - small

- medium
 - large
2. по составительным свойствам с другими цепями:
 - tri
 - trireg
 - tri0, tri1
 - supply0, supply1
 - wor, wand
 - trior, triand
 3. по формату.
 - бит;
 - вектор;
 - массив.

Регистры делятся:

1. по формату;
 - бит;
 - вектор;
 - массив.
2. по типу хранимых данных:
 - двоичные (**reg**);
 - двоичные со знаком (**reg signed**);
 - знаковое целое, 32 бита (**integer**)
 - число с плавающей запятой, размер и формат не фиксирован (**real**)
 - значение времени 64 бита (**time**)

Еще одним типом объектов в Verilog-е являются именованные события – **event**, которые представляют собой абстракцию событий в цифровой схеме – перепадов сигналов, присвоения им определенного значения и т.п. В Verilog события используются в качестве условий в операторах ветвления, при вызове процедурных блоков. Но для удобства можно именовать события определенного типа (то есть создавать объекты **event**) и «вызывать» их по идентификатору.

Объекты-события имеют значения истина – ложь. Применение объектов **event** похоже на использование логических переменных в языках программирования.

Объекты **event** являются синтезируемыми. Однако в цифровой схеме обычно реализуются неявно – не имеют однозначно соответствующего элемента.

2.4 Подключение модуля.

Как говорилось выше, включение модуля как компонента в проект более сложной цифровой системы и соединение с другими модулями осуществляется посредством внешних сигналов модуля - портов.

Тип внешних сигналов и тип портов, к которым эти сигналы подключаются, связаны следующими правилами: **in**-порт подключается к сигналу типа **net** или **reg**; **out**-порт – только к **net**-сигналу, **inout**-порт – только к **net**-сигналу.

При включении модуля в проект, список подключаемых к портам внешних сигналов должен соответствовать списку портов, к которым эти сигналы подключаются. Соответствие устанавливается двумя способами:

1. Перечисление внешних сигналов в том же порядке, как перечислены порты в декларации, например:

```
module Processor (Clock, Reset, Address, Data, Read_Write)
```

```
...
```

```
endmodule
```

```
module Computer
```

```
...
```

```
wire sysclk, sysreset, sysrw;
```

```
wire [19:0] sysadr;
```

```
wire [15:0] sysdata;
```

```
...
```

```
Processor Proc1 (sysclk, sysres, sysadr, sysdata, sysrw);
```

```
endmodule
```

2. Установка связи по именам, например:

```
module Processor (Clock, Reset, Address, Data, Read_Write)
```

```
...
```

```
endmodule
```

```
module Computer
```

```
...
```

```
wire sysclk, sysreset, sysrw;
```

```
wire [19:0] sysadr;
```

```
wire [15:0] sysdata;
```

```
...
```

```
Processor Proc1 (.Clock(sysclk), .Reset(sysres), .Address(sysadr), .Data(sysdata),  
.Read_Write(sysrw));
```

```
endmodule
```

2.4.1 Использование параметров.

Параметры в Verilog – это настройки, конфигурирующие модуль на этапе компиляции. Обычно через параметры задаются задержки или ширина шин, могут включаться или отключаться подсистемы и функции модуля. Параметры не соответствуют никаким структурным элементам (регистрам, сигналам) цифровой системы. Параметры - это константы, которые могут быть использованы в выражениях. Особенность параметров по сравнению с классическими константами в том, что при подключении модуля в систему параметрам этого модуля могут быть присвоены другие значения, чем по умолчанию. Это позволяет конфигурировать модули при «сборке» системы: выбирать задержки, размерность шин. В систему могут быть включены несколько одинаковых модулей, но с различным параметрами.

Новые значения параметрам могут присваиваться только в процессе компиляции («сборки») системы, а в процессе симуляции параметры остаются константами.

Декларация параметров может иметь представленный ниже вид. В выражениях могут использоваться числа, арифметические и логические операции, могут применяться ранее определенные параметры.

```
parameter lsb = 7 ;
```

```
parameter size = 8 , word = 32 ;
```

```
parameter number = 3.92, frequency = 100 ;  
parameter clk_cycle = frequency / 2 ;
```

Ниже показан пример декларации и использования параметров в модуле. Здесь: width – ширина шины, delay – задержка распространения сигнала.

```
module d_trigger (Clk, D, Q) ;  
parameter width = 2, delay = 10 ;  
input [width - 1 : 0] D ;  
input Clk ;  
output [width : 0] Q ;  
assign #delay Q = D;  
endmodule
```

Далее приведен первый способ изменения значения параметра с помощью выражения **defparam**. После **defparam** должен быть указан иерархический путь от места вызова **defparam** до конфигурируемого модуля и его параметра, а после этого должно быть присвоено новое значение параметра. При использовании данного метода группа или все параметры системы могут быть назначены в одном месте. Что удобно при конфигурировании проекта с большим числом иерархических уровней.

```
module top;  
reg Clk ;  
reg [7:0] D ;  
wire [7:0] Q ;  
d_trigger inst_1(Clk, D, Q) ;  
endmodule
```

```
module override ;  
defparam top.inst_1.width = 7 ;  
endmodule
```

Другой вариант изменения значения параметра – непосредственно при инсталляции модуля. При инсталляции модуля в скобках перечисляются значения всех параметров данного модуля. Нельзя пропускать ни один параметр. Такая форма удобна, если не предполагается реконфигурировать модуль в процессе отладки или тестирования системы: например, если в системе использован 8-разрядный регистр, то надо назначить параметр width=8 один раз и не выносить его в «голову» проекта. Пример назначения параметров при инсталляции приведен ниже.

```
module top;  
reg Clk ;  
reg [7:0] D ;  
wire [7:0] Q ;  
d_trigger #(7, 25) inst_1(Clk, D, Q) ;  
endmodule
```

2.5 Построение вентиляных моделей (Gate-level modeling) и примитивы.

Уровень логических элементов или вентилях (GATE LEVEL) является вторым снизу уровнем иерархии, на котором цифровая система рассматривается как комбинационная логическая схема из элементов И, ИЛИ, НЕ, И-НЕ и т.д.

Язык VERILOG позволяет строить текстовые описания вентиляного уровня. Для большинства разработчиков и проектов вентиляный уровень рассматривается нижним для Verilog-a (Однако, следует отметить, что Verilog поддерживает и схемный уровень (circuit-, transistor-, switch –level), для чего имеются примитивы rmos, pmos, grmos, gnmos и другие.) Для построения gate-описаний используются встроенные в язык модули **and**, **or**, **nor** и т. д. Такие встроенные модули называются *примитивами*, так как они находятся на нижнем уровне структурной иерархии, то есть не раскладываются далее на submodule и операции. Функциональность встроенных в язык примитивов обеспечивает простейшие операции, определяемые стандартом языка. Имеется возможность также создавать новые – определяемые пользователем примитивы (User Defined Primitives, UDP).

Встроенные примитивы используются аналогично обычным модулям с той лишь разницей, что в тексте программы не имеется декларации и описания архитектуры этих модулей. При использовании примитивов допускается не указывать имя устанавливаемого элемента, а только тип примитива. Например:

```
wire OUT1, OUT2, OUT3, IN1, IN2; //описание цепей
```

```
//подключение в схему логических элементов
```

```
and a1(OUT1, IN1, IN2); //двухвходового элемента
```

```
nand na1(OUT2, IN1, IN2); //трехвходового элемента
```

```
and (OUT3, IN1, IN2); //не указано имя элемента, а только тип примитива
```

1.1.1 Предопределенные примитивы Verilog.

Любые логические схемы могут быть разработаны используя базовый набор логических элементов (вентилей). В Verilog-е базовый набор включает два класса вентилях: and/or и buf/not.

Элементы класса and/or имеют несколько битовых входов и один битовый выход. В списке портов первым идет выход, далее все входы. Количество входов у элемента определяется автоматически по количеству перечисленных входных сигналов. Verilog имеет следующие встроенные примитивы данного класса: and, or, xor, nand, nor, pxor. (пример см. выше).

Элементы класса buf/not имеют один битовый вход и несколько битовых выходов. В списке портов сначала перечисляются все входы, а выход идет последним. Кроме того имеются модификации примитивов с управляющим сигналом ctrl: когда этот сигнал в активном состоянии примитив работает как обычно, когда сигнал ctrl в неактивном состоянии, то выход имеет высокоомное значение вне зависимости от состояния входов. При неопределенном состоянии на входе управления выход элемента будет в некотором предопределенном состоянии. Сигнал управления будет последним в списке портов, после входа. В Verilog-е поддерживаются несколько примитивов данного класса: простые: buf (повторитель) и not(инвертор), и с управлением: bufif0, notif0 (активный уровень для ctrl = «0»), bufif1, notif1 (активный уровень для ctrl = «1»).

Примеры использования примитивов buf/not:

not n1 (out, in);

buf (out, in);

bufif1 b1 (out, in, ctrl);

Допускается использовать логические примитивы для построения многоразрядных схем.

Например:

```
wire [7:0] OUT, IN1, IN2;
```

```
// инсталляция логических ячеек для операции с многоразрядными векторами.
nand n_gate[7:0](OUT, IN1, IN2);
```

```
// Вышепредставленная запись эквивалентна представленному ниже блоку
nand n_gate0(OUT[0], IN1[0], IN2[0]);
nand n_gate1(OUT[1], IN1[1], IN2[1]);
nand n_gate2(OUT[2], IN1[2], IN2[2]);
nand n_gate3(OUT[3], IN1[3], IN2[3]);
nand n_gate4(OUT[4], IN1[4], IN2[4]);
nand n_gate5(OUT[5], IN1[5], IN2[5]);
nand n_gate6(OUT[6], IN1[6], IN2[6]);
nand n_gate7(OUT[7], IN1[7], IN2[7]);
```

Ниже представлен пример GATE-описания четырехходового мультиплексора.

```
//Модуль мультиплексора 4-в-1
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

//Декларация портов
output out;
input i0, i1, i2, i3;
input s1, s0;

// Декларация внутренних цепей
wire s1n, s0n;
wire y0, y1, y2, y3;

// Инсталляция ячеек

// Формирование сигналов s1n и s0n
not (s1n, s1);
not (s0n, s0);

and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);

or (out, y0, y1, y2, y3);

endmodule
```

1.1.2 Примитивы, определяемые пользователем (UDP).
(см. лекции)

1.1.3 Подпрограммы в Verilog: задачи (TASK) и функции (FUNCTION).

Язык VerilogHDL поддерживает концепцию иерархической организации как для структурных моделей, так и для поведенческих моделей. Структурная иерархия обеспечивается использованием модулей (module) и примитивов (primitive), описание и использование которых описано выше. Поведенческая иерархия поддерживается использованием подпрограмм. Подпрограммы оформляют поведенческие функциональные блоки, которые могут быть использованы в разных частях модели (Verilog-программы). Verilog поддерживает два типа подпрограмм: задачи (task) и функции (function).

Функция – поведенческий блок, вычисляющий некоторое значение по заданному алгоритму и на основании значений входных аргументов. Функция имеет только входные аргументы, результат вычислений возвращается как значение функции. Функция не имеет времени исполнения – в процессе симуляции она исполняется мгновенно, за время 0. Соответственно внутренний алгоритм не может включать задержек и обработки событий. Функции могут вызывать другие функции, но не могут вызывать задачи.

Задача – поведенческая модель некоторой подсистемы, имеет входные и выходные аргументы (можно рассматривать как аналог портов модуля), исполняется во времени: внутренние операции могут исполняться с задержками, допускается обработка событий (event). Задача может вызывать как другие задачи, так и функции.

Обычно одну и ту же задачу или функцию нельзя одновременно вызывать (запускать) несколько раз, так как эти вызовы будут конфликтовать из-за использования одних и тех же переменных. Однако язык Verilog допускает описывать задачи и функции как реентерабельные или рекурсивные, для чего в строке декларации указывается слово automatic. В данном случае будет создаваться отдельное операционное пространство для каждой задачи/функции с индивидуальным набором переменных.

Функции возвращают по умолчанию беззнаковые значения. Чтобы функция возвращала значения со знаком необходимо указать служебное слово signed.

Ниже приводится в формальном виде правила декларации задач и функций, а также примеры:

```
//Формальное описание задачи
task_declaration ::=
    task [ automatic ] task_identifier ;
    { task_item_declaration }
    statement
    endtask
| task [ automatic ] task_identifier ( task_port_list ) ;
  { block_item_declaration }
  statement
  endtask

task_item_declaration ::=
    block_item_declaration
  | { attribute_instance } tf_input_declaration ;
  | { attribute_instance } tf_output_declaration ;
  | { attribute_instance } tf_inout_declaration ;
task_port_list ::= task_port_item { , task_port_item }
task_port_item ::=
    { attribute_instance } tf_input_declaration
  | { attribute_instance } tf_output_declaration
  | { attribute_instance } tf_inout_declaration
tf_input_declaration ::=
    input [ reg ] [ signed ] [ range ] list_of_port_identifiers
  | input [ task_port_type ] list_of_port_identifiers
tf_output_declaration ::=
```

```
        output [ reg ] [ signed ] [ range ] list_of_port_identifiers
        | output [ task_port_type ] list_of_port_identifiers
tf_inout_declaration ::=
        inout [ reg ] [ signed ] [ range ] list_of_port_identifiers
        | inout [ task_port_type ] list_of_port_identifiers
task_port_type ::=
        time | real | realtime | integer
```

```
//Пример модуля, использующего две задачи: инициализационную
//и генерации ассиметричной последовательности
```

```
module sequence;
...
reg clock;
...
initial
    init_sequence; //Вызов задачи init_sequence
...
always
begin
    asymmetric_sequence; //Вызов задачи asymmetric_sequence
end
...
...
//Описание задачи инициализации
task init_sequence;
begin
    clock = 1'b0;
end
endtask

//Описание задачи генерации ассиметричной последовательности
task asymmetric_sequence;
begin
    #12 clock = 1'b0;
    #5 clock = 1'b1;
    #3 clock = 1'b0;
    #10 clock = 1'b1;
end
endtask
...
...
endmodule
```

```
//Формальное описание функции
```

```
function_declaration ::=
    function [ automatic ] [ signed ] [ range_or_type ]
    function_identifier ;
    function_item_declaration { function_item_declaration }
    function_statement
    endfunction
| function [ automatic ] [ signed ] [ range_or_type ]
    function_identifier (function_port_list ) ;
    block_item_declaration { block_item_declaration }
    function_statement
    endfunction
function_item_declaration ::=
    block_item_declaration
```

```

    | tf_input_declaration ;
function_port_list ::= { attribute_instance } tf_input_declaration {,
                        { attribute_instance } tf_input_declaration }
range_or_type ::= range | integer | real | realtime | time

//Пример модуля, использующего функцию calc_parity вычисления бита четности
module parity;
...
reg [31:0] addr;
reg parity;

//Расчет бита четности
always @(addr)
begin
    parity = calc_parity(addr); //Вызов функции вычисления бита четности
    $display("Parity calculated = %b", calc_parity(addr) );
                                //Повторный вызов функции
end
...
...
//Описание функции вычисления бита четности
function calc_parity;
input [31:0] address;
begin
    calc_parity = ^address; //Return the xor of all address bits.
end
endfunction
...
...
endmodule

```

2.6 Системные задачи (system task).

Стандарт Verilog регламентирует использование специального типа задач управления процессом симуляции и отладки – системных. Данные задачи не влияют на функционирование цифровой системы, но позволяют управлять процессом симуляции, осуществлять мониторинг изменения состояния сигналов, вывод и протоколирование в файле состояний сигналов и событий. Стандарт Verilog описывает минимальный – predetermined набор системных задач. Разработчики симуляции могут добавлять собственные системные задачи, описываемые в документации к симуляторам. Правила использования системных задач похожи на обычные задачи: они могут вызываться только в поведенческих блоках. Формат вызова системных задач также схож с обычными задачами:

\$идентификатор (опциональный список аргументов);

2.6.1 Системная задача \$display отображения данных и надписей.

Задача \$display обеспечивает отображение информации в процессе симуляции.

Формат вызова: \$display(p1, p2, p3, ..., pn);

Аргументами p1, p2, p3, ..., pn могут являться переменные, выражения или строки.

Используются спецификаторы формата вывода значений:

Спецификации вывода	
Спецификатор	Описание
%d or %D	Вывод переменной в десятичном формате
%b or %B	Вывод переменной в двоичном формате
%s or %S	Вывод строки
%h or %H	Вывод переменной в шестнадцатиричном формате
%c or %C	Вывод ASCII символа
%m or %M	Вывод иерархического имени исполняемого модуля
%v or %V	Вывод мощности (strength) цепи (net)
%o or %O	Вывод переменной в восьмиричном формате
%t or %T	Вывести в текущем формате времени
%e or %E	Вывести число real в форме со степенью 10, например 3e10
%f or %F	Вывести число real в виде десятичной дроби, например, 2.13
%g or %G	Вывести число real в наиболее короткой форме из описанных выше

Примеры:

```
$display("Hello Verilog World");
-- Hello Verilog World
```

```
reg [0:40] virtual_addr;
$display("At time %d virtual address is %h", $time, virtual_addr);
-- At time 200 virtual address is 1fe0000001c
```

```
reg [4:0] port_id;
$display("ID of the port is %b", port_id);
-- ID of the port is 00101
```

2.6.2 Системная задача \$monitor мониторинга состояния сигналов.

Задача \$monitor обеспечивает индикацию изменения значения сигнала процессе симуляции. Мониторинг аналогичен отображению \$display, но выполняемому по условию изменения значения.

Формат вызова: \$monitor(p1, p2, p3, ..., pn);

Аргументами p1, p2, p3, ..., pn могут являться переменные, выражения или строки. Спецификаторы формата вывода значений аналогичны задаче \$display.

Мониторинг можно выключить и обратно включить с помощью вызовов \$monitoroff и \$monitoron. Для отображения времени возникновения события удобно использовать вызов \$time, возвращающий симуляционное время.

Пример:

```
initial
begin
    $monitor($time,
              " Value of signals clock = %b reset = %b", clock, reset);
```

end

Пример выводимых надписей:

```
-- 0 Value of signals clock = 0 reset = 1
-- 5 Value of signals clock = 1 reset = 1
-- 10 Value of signals clock = 0 reset = 0
```

2.6.3 Системные задачи управления симуляцией.

Наиболее часто используются задачи:

`$stop` – приостановка автоматической симуляции и перевод системы симуляции в интерактивный режим. После этого можно продолжить процесс симуляции с места остановки.

`$finish` – полное прерывание симуляции.

```
initial
begin
clock = 0;
reset = 1;
#100 $stop; // Приостановка симуляции в момент time= 100
#900 $finish; // Прерывание симуляции через время time = 1000
end
```

2.7 Директивы компилятора.

Как и при компиляции с классических языков программирования, Verilog – компиляция может управляться с помощью специальных директив компилятора. Ниже приведены наиболее часто используемые:

1) ``define` - определение выражения

Например: ``define S $stop; //`Вместо вызова `$stop` можно писать `S`

2) ``include` имя_файла //подключение файла

3) ``timescale` единица_измерения_времени / шаг_симуляции

Например: ``timescale 100 ns / 1 ns` - задает значение единицы при определении задержек равное 100 ns, а шаг симуляции = 1 ns.

3 Язык VHDL.

3.1 Общие сведения.

Язык VHDL (Very high speed integrated circuits **H**ardware **D**esign **L**anguage) разработан по инициативе министерства обороны США в начале 1980-х годов на базе языка АДА. В 1987 году был принят стандарт ANSI/IEEE-1076-87 (VHDL'87), в 1993 году – стандарт ANSI/IEEE-1076-93 (VHDL'93). До последнего времени язык VHDL являлся основным международным стандартом в области автоматического проектирования цифровых систем, это входной язык большинства САПР заказных (ASIC) и программируемых (ПЛИС, PLD, CPLD, FPGA, FPLSIC, PSOC и др.) микросхем. В последнее время жесткую конкуренцию языку VHDL составляет VerilogHDL: его тексты более компактные и понятные. Однако VHDL остается широко популярен по причине большого количества наработок и накопленного опыта его использования у многих разработчиков. Кроме этого, VHDL занимает нишу высокоуровневых «системных» описаний, которые недостаточно обеспечиваются средствами VerilogHDL. В настоящее время ведутся работы по слиянию VHDL и Verilog, с целью создания единого стандартного ЯСФО, обеспечивающего простоту использования Verilog и функциональную мощь VHDL.

3.2 Структура VHDL-модели.

Полное описание VHDL-модели блока цифровой системы (формально называется design entity – объект проекта) состоит из двух частей:

1. Декларация объекта и описание внешнего интерфейса (раздел ENTITY), включающее списки входных и выходных сигналов (блок PORT) и список параметров (блок GENERIC).
2. Описание внутренней архитектуры объекта (раздел ARCHITECTURE), включающее объявление переменных, внутренних, сигналов и операторную часть, содержащую функциональное или структурное описание объекта проектирования.

3.2.1 Декларация объекта проектирования и описание интерфейса с внешним окружением.

Раздел **entity** представляет класс объектов (ЦС, блоков ЦС), имеющих одинаковый внешний интерфейс. Интерфейс не имеет неразрывной связи с внутренней реализацией объекта (раздел ARCHITECTURE): один и тот же интерфейс может быть использован различными архитектурами. Это позволяет при проектировании пробовать и оценивать различные варианты реализации ЦС, а так же осуществлять переход от функциональных описаний (одна архитектура), к структурным описаниям (другая архитектура).

Синтаксис:

```
entity entity_name is  
    generic (generic_list);  
    port (port_list);  
end entity entity_name;
```

entity – декларация идентификатора объекта;

generic - список статических параметров (например, задержек, ширины шины), конкретные значения которых могут задаваться при «вызове» объекта – включении в более высокоуровневую структуру. Для каждого параметра задается идентификатор, тип и, факультативно, значение по умолчанию. Описание нескольких параметров перечисляются через запятую.

Пример: **generic** (BusWidth : Integer := 16;
 BusDelay : Time);

port – список сигналов, использующихся как коммуникационные каналы «внутренностей» объекта с его окружением. Порты – всегда объекты **signal**. Для каждого порта задается идентификатор, направление передачи данных, тип и, факультативно, значение по умолчанию. Описание нескольких портов перечисляются через запятую.

Пример:

```
entity Mux8to1 is
  port (
    Inputs      : in Bit_Vector(7 downto 0);
    Select_s    : in Bit_Vector(2 downto 0);
    Output      : out Bit
  );
end Mux8to1;
```

Перед идентификатором порта можно ставить ключевое слово **signal**.

Направления портов могут быть следующие:

- **in** – входной порт. Объектом значение из него может быть прочитано, но не может быть в него записано (назначено).
- **out** – выходной порт. Объектом значение может быть записано (назначено), но не может быть прочитано.
- **inout** – двунаправленный порт. Значение может быть и записано и прочитано.
- **buffer** – выходной порт, но записанное в него значение может быть прочитано. В отличие от inout записывать в него может только один источник «изнутри» объекта. Аналог регистра-защелки.
- **linkage** – значение порта может быть прочитано и записано, но он используется только связанный с интерфейсным объектом.

Тип порта может быть любой, например, float.

3.2.2 Описание внутренней архитектуры объекта.

Архитектура (раздел **architecture**) – «тело» объекта проекта, содержащее функциональное и/или структурное описание объекта проекта ассоциированное с декларацией объекта (разделом entity).

Архитектура всегда связана с одним и только с одним entity-описанием и описывает внутренние зависимости между входами и выходами объекта.

Синтаксис раздела **architecture**:

```
architecture architecture_name of entity_name is
  architecture_declarations
begin
  concurrent_statements
end [ architecture ] [ architecture_name ];
```

Раздел **architecture** состоит из двух частей:

1. декларативная – содержит описание объектов (сигналов, констант, файлов), типов, функций и процедур, компонентов и групп (именованных объектов).
2. операторная – содержит параллельные выражения (concurrent statement), описывающие зависимость между входами и выходами. VHDL поддерживает две формы архитектурных описаний: структурные и функциональные.

Структурное описание описывает объект как совокупность компонентов с зафиксированным интерфейсом и функцией каждого и связей между портами этих компонентов. Компоненты могут быть взяты из библиотек и для проекта считаться

элементарными – неделимыми, или описаны самостоятельно в любом из стилей: структурном или функциональном.

Функциональные описания разделяются на потоковые (dataflow) и поведенческие (behavior). Потоковые описания содержат выражения назначения сигналов (signal assignment statement) представляют сигнал, как функцию от других. Потоковые выражения в архитектурном теле активизируются по изменению значений входных сигналов и выполняются параллельно. На основании потоковых выражений можно представить структуру ЦС, реализующей этот алгоритм. Поведенческие описания, или описания в виде процессов, представляет собой совокупность параллельно исполняющихся процессов, каждый из которых описывает алгоритм определения значения одного или нескольких сигналов, но не дает представления о структуре, реализующей данный алгоритм. Описание алгоритма состоит из последовательности операторов назначения сигналов и условных операторов. В процессах могут использоваться вспомогательные переменные.

Декларативная часть архитектуры.

В декларативной части раздела architecture содержится описания типов, глобальных внутренних объектов (сигналов, констант, общих переменных (shared)), функций и процедур, компонентов, групп. Все описанные типы, объекты и т.д. будут доступны в операторной части архитектуры. В операторной части, например, в процессах, могут быть описаны дополнительные объекты – сигналы, переменные.

3.3 Объекты и типы в VHDL.

3.3.1 Объекты в VHDL.

Объекты – именованные элементы в VHDL-модели, которые имеют определенный тип, им присваиваются значение и они используются для хранения различных данных. Выделено четыре класса объектов:

- константы;
- переменные;
- сигналы;
- файлы.

Константы и переменные характеризуются только текущим значением и не имеют истории. Константам значение задается при создании, и оно неизменно. Значения переменных могут меняться в процессе функционирования. Переменные могут использоваться только в процессах или подпрограммах, хотя декларироваться могут и вне процессов/ подпрограмм для их (процессов) связи. В последнем случае декларация начинается со слова **shared**.

constant имя_константы : тип := значение по умолчанию;

[shared] variable имя_переменной : тип := значение по умолчанию;

Сигналы – специальный тип переменных, которые кроме значения характеризуются историей изменения значений во времени. Сигналы не могут быть декларированы в процессах, так как в процессах «нет времени».

signal имя_сигнала : тип := значение по умолчанию;

Файлы – последовательность объектов одного типа, сохраненная в виде файла в той системе, где производится симуляция. Можно сохранять константы, переменные или

сигналы. С файлами можно выполнять стандартный набор процедур: создать, открыть, закрыть, записать, читать. Файлы не отображаются ни на какую структуру в составе моделируемой ЦС, являются вспомогательными объектами. Декларация файлов может располагаться в `architecture`, `process`, `block`, `package`, `procedure`, `function`.

file имя_файла : тип;

3.3.2 Типы в VHDL.

Каждый объект в VHDL имеет тип. Тип объекта определяет допустимые значения объекта в непрерывном диапазоне или в ряду значений, набор операций, которые можно производить с данным объектом и набор атрибутов объекта.

Декларация типов.

type имя_типа **is** определение_типа;

В VHDL поддерживаются 4 основных класса типов:

- скалярные (`scalar`) – значения данных типов не имеют элементов;
- композитные (`composite`) – значения данных типов состоят из элементов;
- доступа (`access`) – указатель на объекты других типов;
- файлы (`file`) - определяют структуру (составляющие объекты) файлов.

В рамках некоторых классов существуют подклассы, те в свою очередь могут делиться еще на подклассы. В VHDL существуют типы различных классов и подклассов, определенные по умолчанию (в системных библиотеках) – предопределенные типы. При использовании для них не требуется специальной декларации.

3.3.3 Скалярные типы.

Скалярные типы бывают:

- дискретные, в т. ч.:

- целые: **type** имя_типа **is range** n **to** (**downto**) m; где n и m – целые числа. операции: +, -, *, /, mod (деление по модулю, результат – тот же знак, что делитель), rem (остаток), abs (абсолютное значение), ** (экспонента).

предопределенный тип:

Значение по умолчанию для объектов этого типа – левая граница диапазона.

type integer is range - 2147483649 (- $2^{31}+1$) **to** 2147483647 ($2^{31}-1$);

- перечисляемые – в декларации определен список возможных значений, описанных как идентификаторы (отвечают правилам написания идентификаторов) или текстовые строки (в кавычках). Применяется для описания состояний объектов на высоких уровнях абстракции, например, для описания системы команд процессора в мнемониках, для символьного обозначения состояний управляющих автоматов, для описания текущего режима работы системы и др.

Определение:

type имя_типа **is** (значение №0, значение №1, значение №2, ...);

где «значение №x» - символ, строка или идентификатор.

Значение по умолчанию для объектов – левое в списке (№0).

Операции определяются в зависимости от конкретного типа.

Для различных перечисляемых типов допустимы одинаковые значения, напр.:

type logic_level **is** (unknown, low, high);
type water_level **is** (low, normal, high);

Предопределенные перечисляемые типы:

type character **is** (список 256 символов (VHDL87 – 128 символов) – подмножества набора ISO);

type boolean **is** (false, true); Операции: для операций отношения =, /=, >, >=, <, <= - результат Boolean; для логических операций and, or, nand, nor, xor, xnor(нет в VHDL-87), not – операнды и результат Boolean.

type bit **is** ('0', '1'); Операции: логические - and, or, nand, nor, xor, xnor(нет в VHDL-87), not – операнды и результат Bit. Тип бит используется для описания аппаратуры, boolean – для абстрактных моделей.

Предопределенные перечисляемые типы библиотеки ieee.std_logic_1164. Для подключения к проекту требуется объявление:

library ieee;

use ieee.std_logic_1164.all;

type std_ulogic **is** ('U', -- неинициализированный
'X', --неопределенный
'0', -- ноль
'1', -- единица
'Z', -- высокоомное состояние
'W', -- слабый неопределенный
'L', --слабый ноль
'H', -- слабая единица
'-'); -- безразличный

type std_logic **is** (...); - resolved тип – допускает несколько источников.

Операции: логические.

- с плавающей точкой: **type** имя_типа **is range** n **to(downto)** m; где n и m – дробные числа.

операции: +, -, *, /, abs (абсолютное значение), ** (экспонента).

предопределенный тип:

Значение по умолчанию для объектов этого типа – левая граница диапазона.

type real **is range** - 1E38 **to** 1E38;

- физические – для определения физических величин: длины, массы, времени, тока и др. Кроме значения имеют единицы измерения. Единицы измерения бывают первичные (основные, primary) и вторичные (производные, secondary). Декларация физических типов:

type имя_типа **is range** n **to(downto)** m

units

имя_перв_единицы;

[имя_втор_единицы1 = целый_множитель имя_перв_единицы;

имя_втор_единицы2 = целый_множитель имя_перв_единицы;

.....]

end units [имя_типа]; где n, m – целые или дробные.

операции: такие же, как у целых типов или с плавающей точкой, в зависимости от описания границ диапазона физического типа.

предопределенный тип:

type time **is range** n **to** m

-- n, m – зависят от реализации.

units

fs;

ps = 1000 fs;

ns = 1000 ps;

us = 1000 ns;

ms = 1000 us;

sec = 1000 ms;

min = 60 sec;

hr = 60 min;

end units;

3.3.4 Субтипы.

Субтип – определяется как подмножество базового типа с суженным диапазоном значений (процедура `constraint` – сужение, ограничение). Субтипы удобно использовать для ограничения диапазона значений для группы объектов в проекте, изменения направления диапазона (`to` на `downto` или наоборот). Объекты определенного субтипа поддерживают те же операции, что и базовый тип, им могут присваиваться значения базового типа и наоборот: без преобразования типов могут использоваться вместе с объектами базового типа в операциях.

Декларация субтипа:

subtype имя_субтипа **is** имя_базового_типа **range** n **to**(**downto**) m;

Предопределенные субтипы:

subtype natural **is** integer **range** 0 **to** 2147483647;

subtype positive **is** integer **range** 1 **to** 2147483647;

subtype delay_lenght **is** time **range** 0 fs **to** макс_значение_time;

3.3.5 Преобразование типов.

Функции преобразования используются, чтобы присваивать значения одних типов объектам других типов. Например: преобразование численных значений возможно для типов `integer` и `real`, например, `integer (3.6); real (123);`

3.3.6 Составные (композиционные) типы.

Объекты составных типов представляют собой структуру из других объектов, называемых элементами. Элементы могут быть скалярного, составного типов или типа `access`. Не допускается использование элементов типа `file`.

Составные типы VHDL разделяются на два класса: массивы (`array`) и записи (`record`). Массив состоит из элементов одинакового типа. Элементы записей могут иметь (но не обязательно) различный тип.

3.3.6.1 Массивы.

Массив – класс составных типов, который определяет объекты, состоящие из нескольких элементов одинакового типа. Каждый элемент определяется в массиве индексом. Индекс должен быть дискретного скалярного типа. Индекс может быть или целым числом или значением перечисляемого типа, например, `character`, `boolean`, `bit`.

Декларация массивов:

type имя_типа **is array** (дискретный_диапазон_индекса) **of** тип_элементов_массива;

Например:

type word **is array** (0 to 31) **of** bit;

type word **is array** (0 **downto** 31) **of** bit;

При использовании в качестве индекса объекта перечисляемого типа диапазон индекса должен быть непрерывным:

type state **is** (init, idle, active, error);

type state_counts **is array** (idle **to** error) **of** natural;

Если несколько различных перечисляемых типов могут принимать одинаковые значения, например, (idle to error), то необходимо явно указать тип индекса:

type state_counts **is array** (state **range** idle **to** error) **of** natural;

Для индексов любых типов можно описывать диапазон индекса в виде субтипов:

subtype state_cnt **is** state **range** idle **to** error;

type state_count **is** (state_cnt) **of** natural;

3.3.6.2 Многомерные массивы.

В данном случае используются несколько диапазонов индексов (два и более):

type symbol **is** ('a', 't', 'd', 'h', digit, cr, error);

type state **is range** 0 **to** 3;

type tr_matrix **is array** (state, symbol) **of** state;

type convert **is array** (1 **to** 10, 1 **to** 10) **of** integer;

Значения элементов массивов могут быть назначены по одному, а могут сразу для многих элементов. Это можно делать при инициализации или в операциях присваивания:

type status **is array** (0 to 3) **of** integer;

variable st_sample : status := (10, 20, 30, 30);

variable st_sample : status := (10, 20, **others** =>30);

variable st_sample : status := (0=>10, 1=>20, 2 **to** 3 =>30);

variable st_sample : status := (0=>10, 1=>20, **others** =>30);

type convert **is array** (1 **to** 10, 1 **to** 10) **of** integer;

constant con_cnt : convert := (

1 => (1=>1, 2=>4, **others** =>0),

2 => (1=>0, 2=>5, **others** =>0),

others => (1 **to** 10 => 1));

Неограниченными массивами (Unconstrained Array types) называют типы-массивы, для которых диапазон индекса не задается при декларации типа. Тогда определение числа элементов массива происходит при декларации конкретного объекта одним из нескольких способов. Например:

type sample **is array** (natural **range** <>) **of** integer;

variable sample_var : sample (0 to 63);

subtype sample_sub **is** sample (0 to 63);

variable sample_var : sample_sub;

constant sample_const : sample := (0=>10, 1=>23, 3=>16, 2=>100, 4=>11);

constant sample_const : sample := (10, 23, 100, 16, 11);

3.3.6.3 Предопределенные типы массивов:

Библиотека STANDART:

type string **is array** (positive **range** <>) **of** character;

type bit_vector **is array** (natural **range** <>) **of** bit;

Библиотека STD_LOGIC_1164:

type std_ulogic_vector **is array** (natural **range** <>) **of** std_ulogic;

3.3.6.4 Операции с объектами-массивами:

- С элементами массива применимы все операции для объектов базового типа (типа элементов);
- с целыми массивами:
 - o с одномерными массивами элементов bit или Boolean: and, or, nand, nor, xor, xnor, not, sll, srl, sla, sra, rol, ror.
 - o с одномерными массивами различных типов:
 - сравнение*: $a < b$, если:
 - a имеет нулевую длину, a b – нет;
 - или $b(i) > a(i)$ при $b(k)=a(k)$, $k < i$.
 - сцепление*: $a \& b$, например, “abc” & ‘d’ = “abcd”.

3.3.6.5 Записи (Record)

– композитный тип, состоящий из элементов различных типов, в том числе композитных.

Декларация:

type имя_типа **is record**

имя_поля1 : тип;

имя_поля2 : тип;

.....

имя_поляМ, имя_поляК : тип;

имя_поляN : тип;

end record имя_типа;

Например:

type time_stamp **is record**

seconds : integer range 0 to 59;

minutes : integer range 0 to 59;

hours : integer range 0 to 23;

end record имя_типа;

Присваивание может быть одному элементу, между записями и агрегатами.

```
variable sample_time, current_time : time_stamp;  
.....  
current_time := (2, 3, 4);  
current_time := (2, others => 4);  
current_time.seconds := 1;  
sample_time := current_time;
```

3.3.7 Тип access (доступ)

Тип *access* указывает и позволяет обращаться к динамически создаваемым объектам других типов. Объектом типа *access* может быть только переменная, но не сигнал или другой. Значение доступа, присваиваемое переменной типа *access*, выдает специальная программа *allocator*, создающая в памяти динамический объект. После присваивания значения, выданного *allocator*-ом *access*-переменной, к объекту можно обращаться по имени *access*-переменной. Значение по умолчанию для типа *access* – **null**.

```
type test_record is record  
    test_time : time;  
    test_value : Bit_Vector (0 to 3);  
end record test_record;  
type AccTR is access test_record;    --access type  
variable x,z : AccTR;              -- access type variable  
...  
z, x := new test_record;           --allocator  
z.test_time := 30 ns;  
z.test_value := B"1100";
```

3.3.8 Атрибуты.

Атрибутом называют константу, сигнал, переменную, тип, диапазон (range) или функцию связанную с одним или несколькими именованными элементами VHDL-описания. В качестве таких элементов рассматриваются типы, субтипы, объекты (сигналы, переменные, константы, файлы, объекты access), функции, подпрограммы, объекты ЦС (entity), архитектуры, метки, компоненты, литералы, группы, пакеты. Атрибуты предоставляют дополнительную информацию об элементе VHDL-описания.

Часть атрибутов предопределена стандартом VHDL, но также могут быть использованы атрибуты, определяемые пользователем.

Сами по себе атрибуты так же являются объектами определенных типов и имеют соответствующее типу значение. При этом тип атрибута и тип (тип объекта), к которому этот атрибут относится, могут быть разными.

Атрибуты имеют следующую форму записи:

Форма записи идентификатора атрибута:

$T'A(X)$ где T – идентификатор типа или объекта;

A – идентификатор атрибута;

X – аргумент, определен только для некоторых атрибутов.

3.3.8.1 Предопределенные атрибуты скалярных типов.

$T'left$ – первое(левое) значение в диапазоне значений при определении типа;

$T'right$ – последнее(правое) значение в диапазоне значений при определении типа;

$T'low$ – наименьшее возможное значение;

$T'high$ – наибольшее возможное значение;

$T'ascending$ – атрибут булевого типа, направление диапазона: true – возрастает, false – уменьшается;

$T'image(x)$ – атрибут типа символьная строка, строковое представление значения x ;

$T'value(s)$ – значение типа T , которое записано в строке s .

Для скалярных дискретных типов (целых и перечисляемых) имеется дополнительный набор предопределенных атрибутов:

$T'Pos(s)$ – номер значения s в типе T .

$T'Val(x)$ – значение позиции номер x в типе T .

$T'Succ(s)$ – значение в позиции s с номером, большим на 1, чем номер позиции со значением s .

$T'Pred(s)$ – значение в позиции s с номером, меньшим на 1, чем номер позиции со значением s .

$T'Leftof(s)$ – значение в позиции слева от позиции со значением s .

$T'Rightof(s)$ – значение в позиции справа от позиции со значением s .

3.3.8.2 Предопределенные атрибуты массивов:

$A'left(N)$ – левая граница индекса для N -го измерения массива типа A ;

$A'right(N)$ – правая граница индекса для N -го измерения массива типа A ;

$A'low(N)$ – минимальное значение индекса для N -го измерения массива типа A ;

$A'high(N)$ – максимальное значение индекса для N -го измерения массива типа A ;

$A'range(N)$ – диапазон индекса для N -го измерения массива типа A ;

$A'reverse_range(N)$ – обратный диапазон индекса для N -го измерения массива типа A ;

$A'length(N)$ – длина диапазона индекса для N -го измерения массива типа A ;

$A'ascending(N)$ – истина, если диапазон индекса для N -го измерения массива типа A ;

3.3.8.3 Атрибуты определяемые пользователем.

В VHDL допускается создавать новые, определяемые пользователем атрибуты, в которых хранится дополнительная информация об элементах проекта. Для этого сначала

необходимо объявить атрибут, как новый объект в проекте. Декларация выглядит следующим образом:

```
attribute имя_атрибута : тип_атрибута;
```

Примеры:

```
attribute Pin_number : Positive;
```

```
attribute Max_delay: Time;
```

Атрибуты могут иметь любой скалярный тип, исключая типы **access** и **file**, и любой композитный тип, исключая типы с элементами типа **access** и **file**.

Далее должна быть спецификация атрибута. В ней связывается атрибут с конкретным элементом VHDL-модели: с объектом, типом, архитектурой, подпрограммой и т.п.

```
Attribute имя_атрибута of имя_элемента : класс_элемента is выражение;
```

Примеры:

```
attribute Component_symbol of Comp_1: component is "Counter_16";
```

```
attribute Coordinate of Comp_1: component is (0.0, 17.5);
```

```
attribute Pin_code of Sig_1: signal is 17;
```

```
attribute Max_delay of Const_1: constant is 10 ns;
```

3.3.8.4 Атрибуты сигналов.

Атрибуты сигналов – показывают историю изменения сигнала и характеристики прошедших назначений (транзакций) и изменений.

S'delayed (T) – сигнал S задержанный на время T;

S'quiet (T) – атрибут типа **boolean**, истина, если в течении времени T не было операций назначения сигнала S (даже без изменения состояния);

S'transaction - атрибут типа **bit**, изменяет свое значение на противоположное при любой операции назначения сигнала S;

S'event - атрибут типа **boolean**, истина, если в течении последнего цикла симуляции было изменение значения сигнала S;

S'active - атрибут типа **boolean**, истина, если в течении последнего цикла симуляции была операция назначения сигнала S;

S'last_event – атрибут типа **time**, время от последнего изменения значения сигнала;

S'last_active – атрибут типа **time**, время от последней операции назначения сигнала;

S'last_value – значение сигнала перед последним изменением этого значения.

3.4 Выражения, операторы и симуляция VHDL-модели.

Одно из основных назначений ЯСФО-модели (кроме формального описания и синтеза ЦС) является симуляция функционирования ЦС с целью функциональной (поведенческой) и временной верификации модели, обнаружение ошибок в структуре и поведении ЦС. Симуляция подразумевает моделирование поведения ЦС. Поведение ЦС описывается с помощью выражений различного типа (например, выражений

присваивания, процессов). Каждое выражение имеет индивидуальный алгоритм исполнения и одновременно подчиняется некоторым единым правилам симуляции. Данные правила имеют допущения, отличающие симуляцию от законов функционирования реальной электронной аппаратуры. Чтобы построить модель наиболее близкую к реальной ЦС, необходимо представлять общие правила симуляции и индивидуальные для каждого оператора.

3.4.1 Общие правила симуляции.

Симуляция модели на языке VHDL (как впрочем и на языке Verilog) выполняется в три стадии. Первые две стадии – подготовительные. Третья – собственно симуляция.

1. Анализ (analysis). На этом этапе выполняется синтаксическая и семантическая (смысловая) проверка программы на ЯСФО. Синтаксис проверяется для каждого выражения в соответствии с грамматическими правилами языка. Для анализа семантики необходимо комплексно анализировать элементы модели (entity, Architecture) или модель целиком.
2. Разборка (elaboration). Разборка подразумевает раскладывание модели на составляющие ее поведенческие описания. Сначала в описании первого уровня иерархии инсталлированные компоненты заменяются их внутренним описанием. На следующем этапе компоненты со второго уровня иерархии заменяются их внутренним описанием взятым с третьего уровня. Процесс разбора продолжается циклически, спускаясь по уровням иерархии, пока модель не будет состоять только из функциональных (поведенческих и потоковых) элементов. После этого она может быть симулирована подобно программе, так как представляет собой описание алгоритма.
3. Исполнение (execution). Непрерывное время функционирования реальной системы в VHDL разбито на дискретные шаги по 1fs. Время возникновения любых событий в системе кратно этой дискрете. Это называется «дискретная событийная модель симуляции». В рамках данной модели реакция на какое-либо событие реальной системы не может быть мгновенной. Она может произойти только позже, через один или несколько временных квантов (дискрет). Например, в процессе нельзя изменить и тут же проанализировать сигнал. Для этого нужно использовать переменные – виртуальный объект не имеющий истории. В ЯСФО события могут быть «назначены» не только на следующий временной квант, но и через некоторое время. Такая модель называется «назначение (планирование) транзакций» (scheduling a transaction).

Симуляция имеет фазу инициализации, на которой выполняются присваивания по умолчанию и запускаются все процессы.

В процессе симуляции все выражения в теле архитектуры (см. ниже) выполняются на каждом цикле симуляции (каждую фемтосекунду) параллельно, используя в качестве аргументов значения, полученные на предыдущем цикле симуляции. Особым типом выражений являются процессы. Все последовательные операторы каждого процесса выполняются в цикле на каждом цикле симуляции, пока не встретится оператор WAIT или, если используется список чувствительности процесса, пока не будет выполнен последний оператор в теле процесса. Подробнее о процессах смотри ниже.

3.4.2 Структура операторной части объекта.

Операторная часть раздела architecture содержит параллельные выражения (concurrent statement), описывающие зависимость между входами и выходами. VHDL поддерживает две формы архитектурных описаний: структурные и функциональные, среди последних выделяются потоковые (dataflow) и поведенческие (behavior). Все типы описаний могут

присутствовать в теле архитектуры одновременно. Таким образом, операторная часть будет включать следующие типы параллельных выражений (concurrent statement):

architecture имя **of** имя_entity **is**

 блок деклараций

begin

Concurrent statements:

- потоковые выражения (dataflow statements);
- процессы (processes);
- параллельные выражения назначения сигналов (concurrent signal assignment statement);
- параллельные утверждения (concurrent assertion statement);
- подключение компонентов (component installation);

end architecture имя;

Здесь: *dataflow statements* – потоковое описание; *process* – поведенческое описание; *concurrent signal assignment statement*, *concurrent assertion statement* – стандартизованные типы процессов с сокращенным синтаксисом, так же относятся к поведенческим описаниям; *component installation* – структурное описание.

3.4.3 Потоковые выражения и задержки.

В потоковых выражениях (dataflow statements) – присваиваются значения сигналам (signal assignment statement). Потоковые выражения непосредственно в теле архитектуры активизируются по изменению значений сигналов–аргументов, и все выражения выполняются параллельно. Потоковые выражения могут использоваться в процессах, но в данном случае они будут исполняться только при активизации процесса и в соответствии с его внутренним алгоритмом.

[label:] имя_сигнала <= [тип задержки] выражение [**after** задержка], [выражение **after** задержка, ...]; где label – метка, выражение – фиксированное значение или выражение;

Для описания задержки назначения сигнала используется оператор **after**, например:

a <= b **and** c **after** 10 ns;

Здесь новое значение сигналу a будет присвоено через 10ns после изменения значения (b **and** c). При этом новое присваиваемое значение будет равно значению (b **and** c) на момент его изменения, т.е. 10ns назад.

Допустима более сложная форма записи:

..

constant tdelay := 10ns;

begin

a<=b **after** tdelay, c **after** 2*tdelay;

В данном случае значение b будет присвоено через 10ns, а значение c – через 2*10ns=20ns. Следует следить, чтобы перечисленные задержки не уменьшались, иначе будет ошибка. Если следующие подряд задержки равны – то присвоится правое значение (при этом при симуляции могут быть показаны пики, некоторые симуляторы могут равенства не допускать). Выражение с несколькими задержками может эффективно использоваться для описания генераторов сигналов (импульсных последовательностей) различной формы:

clock_gen: **process** (clk) **is**

begin

if clk = '0' **then**

 clk <= '1' **after** Tclk, '0' **after** 2*Tclk;

end if;

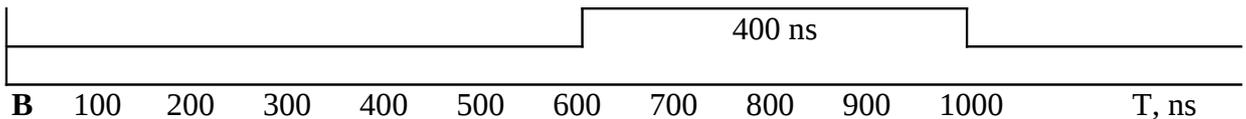
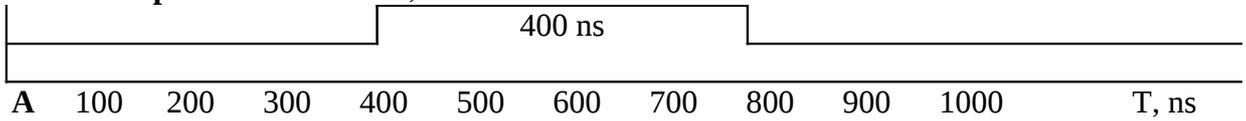
end process clock_en;

Тип задержек:

В VHDL имеется два основных типа задержек:

1. Транспортная – описывает задержку изменения сигнала относительно момента изменения источника. Форма выходного (задержанного) сигнала при этом повторяет форму входного (не задержанного) сигнала.

A <= transport B after 300 ns;



Если назначается несколько транзакций сигнала с увеличивающимся абсолютным временем исполнения, то эти транзакции ставятся в очередь. Например:

B <= transport '1' after 300 ns, '0' after 500 ns, '1' after 800 ns;

A <= transport B after 300 ns;

Здесь для сигнала B назначено три последовательных транзакции и для A три транзакции с задержанным временем исполнения. Если вызов выражения был в 0 ns, то очереди будут следующими:

B: '1' в 300 ns, '0' в 500 ns, '1' в 800 ns.

A: '1' в 600 ns, '0' в 800 ns, '1' в 1100 ns.

Если транзакции назначаются не по возрастанию (асимметрично), то исполняется следующее правило: транзакция с более поздним временем назначения (вызова потокового выражения), но с ранним или равным временем исполнения отменяют все более поздние по времени исполнения транзакции, уже находящиеся в очереди. Например:

DLY: process (a) is

begin

if a = '1' then

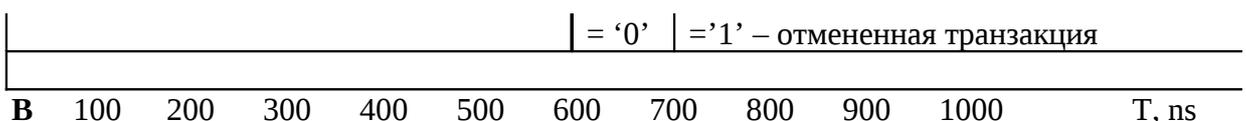
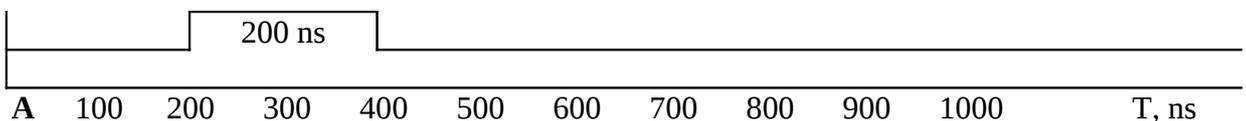
z <= transport a after 500 ns;

else

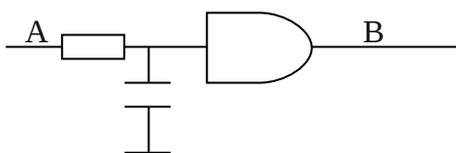
z <= transport a after 200 ns;

end if;

end process DLY;

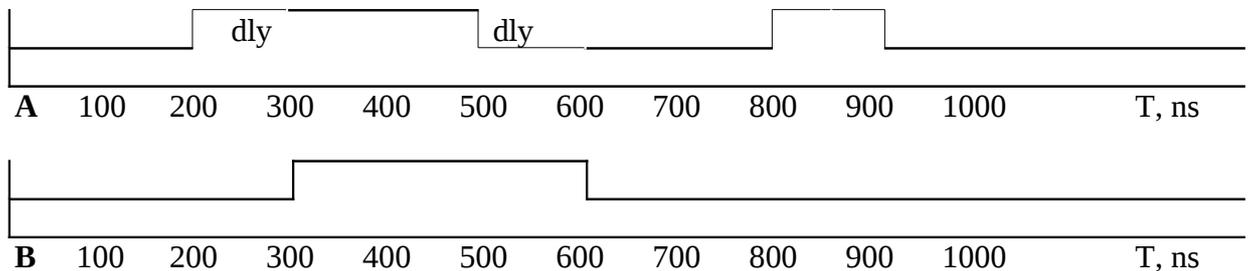


2. Инерционная задержка описывает системы с ограниченной по частоте пропускной способностью. В реальных электрических схемах такие ограничения обусловлены наличием реактивных цепей RC, RL, RLC. Цепь с инерционной задержкой будет



пропускать все импульсы с длительностью большей или равной величине задержки и не пропустит импульсы с длительностью меньшей задержки. Например:

```
dly := 100 ns;
B <= [inertial] A after dly;
```



Полная форма описания инерционной задержки имеет следующий вид:

```
B <= [[reject значение_времени_режекции] inertial] выражение after значение_задержки;
```

Операторы [[**reject** значение_времени_режекции] **inertial**] могут быть опущены (но не может использоваться **reject** без **inertial**). Таким образом, задержка без явного указания типа будет инерционной.

Выражение **reject** используется, чтобы указать минимальную длительность пропускаемого схемой импульса меньшую, чем значение задержки. При этом пропускаемые импульсы могут быть короткими, а задержка этих импульсов при прохождении через схему может быть значительная. Например:

```
B <= reject 10ns inertial A after 30ns;
```

Данная «цепь» пропустит импульсы больше или равные 10ns, но задержит их на 30ns.

Выражение `B <= reject 0ns inertial A after 30ns` будет эквивалентно транспортной задержке, так как пропустит импульсы любой длительности (больше или равной 0). В случае если значение режекции больше, чем значение задержки, то значение режекции будет игнорироваться. Например:

```
B <= reject 40ns inertial A after 10ns; и B <= A after 30ns;
```

выражения эквивалентны, если отсутствуют множественное назначение транзакций (см. ниже).

Как и для транспортных, для инерционных задержек существует очередь транзакций и правила множественного назначения:

- транзакция отменяет все ранее поставленные в очередь транзакции, если они должны быть активизированы одновременно или позже.
- транзакция отменяет другие транзакции, если они должны быть активизированы в течение времени T, перед активизацией данной транзакции, где T – параметр `reject`, и эти транзакции по значению не совпадают с данной транзакцией, а если совпадают, то не удерживают значение в течение остатка периода T.

Например:

```
test_pr: process (clk) is
begin
    S <= '1' after 11ns, '0' after 12ns, '1' after 14ns, '0' after 15 ns,
        '1' after 16ns, '1' after 17ns, '0' after 20ns, '1' after '25ns;
    S <= reject 5ns inertial '1' after 18 ns;
end process test_pr;
```

в очереди были следующие транзакции для сигнала S:

11ns – '1', 12ns – '0', 14ns – '1', 15ns – '0', 16ns – '1', 17ns – '1', 20ns – '0', 25ns – '1'

Было выполнено потоковое выражение:

S <= reject 5ns inertial '1' after 18 ns.

В результате: [11ns – '1', 12ns – '0'] – не попадают в период T и остаются; [14ns – '1'] – удаляется, так как совпадает с новой транзакцией, но не удерживает значения, потому что за ней идет транзакция [15ns – '0']; [15ns – '0'] – удаляется, так как не совпадает по значению с новой транзакцией; [16ns – '1', 17ns – '1'] – остаются, так как совпадают и удерживают значения до новой транзакции; [18ns – '1'] – это сама новая транзакция и она будет поставлена в очередь; [20ns – '0', 25ns – '1'] – удаляются, так как активизируются после новой транзакции.

3.4.4 Процессы (Process).

Процесс – независимо выполняемая последовательность инструкций, описывающая поведение части или всего цифрового устройства.

[метка_процесса:] **process** [(список активности - sensitivity_list)] [**is**]

локальные декларации;

begin

последовательные выражения;

end process [метка_процесса] ;

Исполнение последовательных выражений процесса начинается при старте симуляции и продолжается непрерывно в течение всего времени функционирования модели. Дойдя до последней инструкции, процесс начинает исполняться заново. При таком непрерывном исполнении один «проход» процесса будет выполняться за один цикл симуляции (1 fs (1 фемтосекунда)). Исполнение процесса может быть приостановлено оператором **Wait** на определенное время или до определенного события (изменения сигнала). К заголовку может быть добавлен список активности с перечисленными в нем сигналами, например:

process (A,B);. Такой список эквивалентен оператору **wait on** A, B; ожидающему изменения этих сигналов и расположенному в конце тела процесса. То есть последовательность операторов будет выполняться при каждом изменении этих сигналов. В процессах, использующих список активности, не могут применяться операторы **wait**, ни в теле самого процесса, ни в подпрограммах, вызываемых в процессе.

В декларативной части процесса могут быть объявлены подпрограммы, типы, субтипы, константы, переменные, файлы, атрибуты, объявления использования (USE). В процессах нельзя объявлять сигналы и общие (shared) переменные.

3.4.5 Последовательные операторы.

Последовательные операторы (sequential statement) используются в процессах, процедурах, функциях, выполняются последовательно, друг за другом, с учетом ветвления алгоритма.

ОПЕРАТОР WAIT -

приостанавливает исполнение последовательности операторов, до исполнения некоторого условия. Могут быть несколько вариантов условий:

wait; - остановка исполнения последовательности навсегда;

wait on A,B,...; - остановка исполнения до активности (изменения) сигнала из списка.

wait until булево выражение; - остановка, пока выражение не станет истиной.

wait for время; - остановка на время.

Например:

```
A: process is
    begin
        clk <= '1' after T, '0' after 2*T;
        wait for 2*T;
        (wait until clk = '0';)
    end process A;
```

ОПЕРАТОР **IF** – оператор условного ветвления.

```
if условие (булево выражение) then
    последовательность_операторов;
[[elsif condition then
    последовательность_операторов;]
else
    последовательность_операторов;]
end if;
```

ОПЕРАТОР **CASE** – исполняет одну из последовательностей операторов в зависимости от значения выражения.

```
case expression is
    when choice => sequential_statements
    when choice => sequential_statements
    ...
    [when others => sequential_statements]
end case;
```

when others – в **CASE**-операторе может быть использовано не больше одного раза и только перед **end case**.

Значение choice можно определять множественно (через **ИЛИ**):

```
when A|B => sequential_statements
Можно выражения определять как диапазоны или субтипы:
when A to C => sequential_statements
when C downto A => sequential_statements
when x_subtype => sequential_statements
```

ОПЕРАТОР **NULL** - пустая последовательность.

Вызов: **null;**

Может использоваться на этапе разработки модели для «заглушки» ветвей алгоритма.

ОПЕРАТОР **LOOP** – цикл:

Цикл без условия:

```
метка: loop
    sequence_of_statements
```

end loop метка;

Цикл с предусловием (цикл **while**):

метка: **while** condition **loop**
sequence_of_statements
end loop метка;

Счетный цикл (цикл **for**):

метка: **for** loop_parameter **in** дискретный_диапазон **loop**
sequence_of_statements
end loop метка;

Цикл будет выполнен последовательно для всех значений параметра loop_parameter в указанном дискретном диапазоне. Дискретный диапазон можно определять любым из обозначенных ранее способов (**to**, **downto**, субтипом). Если диапазон пустой, например, 10 **to** 1, то тело цикла не выполнится ни разу.

Параметр loop_parameter декларируется неявно, виден только в рамках цикла и ему не может быть присвоено значение в теле цикла. Существует механизм затенения: если объявлен объект с таким же именем, как и loop_parameter, то вне тела цикла мы будем работать с этим объектом, внутри тела – с loop_parameter, т.е. loop_parameter скрывает декларированный ранее объект. Например:

hiding_exemple: **process is**
variable a,b : integer;

begin
a = 10;
for a **in** 0 **to** 7 **loop**
b:=a;
end loop;
-- a=10, b=7.
end process;

ОПЕРАТОР **NEXT** – завершение текущей итерации цикла **loop**:

next; - следующая итерация в текущем цикле.
next loop_label; - следующая итерация в цикле loop_label при вложенных циклах.
next loop_label **when** условие; - следующая итерация в цикле loop_label запускается при выполнении условия.

ОПЕРАТОР **EXIT** – завершение цикла **loop**:

exit; - завершение текущего цикла.
exit loop_label; - завершение цикла loop_label при вложенных циклах.
exit loop_label **when** условие; - завершение цикла loop_label при выполнении условия.

ВЫРАЖЕНИЕ УТВЕРЖДЕНИЯ **ASSERT**.

Выражения утверждения проверяет условие, и если оно не верно, выдает сообщение об ошибке:

assert условие

[**report** строковое_сообщение]

[**severity** уровень_ошибки(severity_level)];

SEVERITY_LEVEL может быть NOTE, WARNING, ERROR, and FAILURE; по умолчанию (без строки **severity** severity_level) – ERROR.

ВЫРАЖЕНИЕ СООБЩЕНИЯ REPORT

Выражение **report** может использоваться отдельно:

report строковое_сообщение [**severity** уровень_ошибки(severity_level)];

SEVERITY_LEVEL может быть NOTE, WARNING, ERROR, and FAILURE; по умолчанию (без строки **severity**) severity_level – NOTE.

3.4.6 Параллельные выражения назначения сигналов.

Параллельные выражения назначения сигналов (concurrent signal assignment statement) являются сокращенной формой типовых процессов, используются непосредственно «в теле архитектуры» и бывают двух видов:

- a) условное назначение (conditional signal assignment statement) является функциональным аналогом процесса с условным оператором **if**:

```
имя_сигнала <= [тип задержки] выражение_1 [задержка] when условие_1 else
    ...
    выражение_n [задержка] [when условие_n];
```

Для указания неизменности сигнала в любом месте конструкции вместо присваиваемого выражения можно использовать ключевое слово **unaffected**.

Например:

```
z <= d0 when sel1='0' and sel0='0' else
    d1 when sel1='0' and sel0='1' else
    d2 when sel1='1' and sel0='0' else
    unaffected when sel1='1' and sel0='1';
```

- b) назначение по селектору (selected signal assignment statement) является функциональным аналогом процесса с оператором **case**:

with выражение_селектор **select**

```
имя_сигнала <= [тип_задержки] выражение_1 [задержка] when значение_1_селектора,
    ...
    [тип_задержки] выражение_n [задержка] when значение_n_селектора;
```

Например:

with A **select**

```
result <= A+B after 5 ns when 0,
    <= A-B after 10 ns when 1,
    <= A or B after 1 ns when 2;
```

3.4.7 Параллельные выражения утверждения.

Параллельные выражения утверждения (concurrent assertion statement) - являются сокращенной формой процесса с оператором **assert** и используются непосредственно «в теле архитектуры»:

[метка:] **assert** условие

[**report** строковое_сообщение]

[**severity** уровень_ошибки(severity_level)];

SEVERITY_LEVEL может быть NOTE, WARNING, ERROR, and FAILURE; по умолчанию (без строки **severity**) severity_level=ERROR.

В случае, если условие = ложь, симулятором производится обработка ошибки в соответствии с установленным уровнем ошибки и, если указано, выдается строковое сообщение.

3.4.8 Структурное описание цифровых систем. Подключение компонент (component installation).

Структурное описание представляет собой описание множества подсистем (компонент), соединенных между собой сигналами. В свою очередь каждая подсистема может быть описана аналогичным образом – в виде совокупности более простых компонент – и так далее, до уровня примитивов, где внутренняя архитектура компонент представлена в виде поведенческой модели.

Выражения подключения компонент определяют компоненты, включаемые в структуру системы, указывают, какие сигналы будут подключены к портам компонент или какие константные значения будут поданы на входные порты компонент, указывают значения, которые будут присвоены generic-параметрам при подключении компонент.

В VHDL существует три варианта компонент, которые могут быть подключены к структурному описанию:

объекты (entity) - описания объектов (интерфейс с подключенной к нему архитектурой). Внутренняя архитектура подключаемого объекта указывается в самом разделе architecture, или при инсталляции компонента.

```
label : entity entity_name [(architecture_identifier)]
      generic map ( generic_association_list )
      port map ( port_association_list );
```

компоненты (component) – описания интерфейсов не связанные жестко с описанием внутренней архитектуры. Выбор архитектуры происходит в разделе конфигурации (configuration), где указывается, какой объект (entity + architecture) сопоставить каждому подключенному компоненту.

```
label : [ component ] component_name
      generic map ( generic_association_list )
      port map ( port_association_list );
```

конфигурации (configuration) – используются если в разделе конфигурации компонента не совпадают количество и названия портов компонента и сопоставляемого ему объекта. В таком случае подключения используются для переименования портов и подачи на неиспользуемые порты константных значений.

```
label : configuration configuration_name
      generic map ( generic_association_list )
      port map ( port_association_list );
```

При подключении внешние сигналы могут ставиться в соответствие портам компонента двумя способами.

- перечислением по порядку:

```
entity dgblock is
  port ( rd , wr : in bit);
```

end entity dgblock;

dgcomp: **entity** dgblock
port map (xrd, xwr);

- явным указанием соответствия:

...

dgcomp: **entity** dgblock
port map (rd=>xrd, wr=>xwr);

При инсталляции компонент входным портам могут быть назначены не только сигналы, но и выражения, например:

...

port map (a or b);

Сигналы могут быть не использованы, например:

...

port map (a=>x, b=>**open**);