Оглавление

Принятые сокращения 8

Глава 1. Введение 9

1.1. Современные языки проектирования цифровых устройств 9
1.2. Краткая характеристика языка VHDL 10
1.3. Этапы проектирования с использованием VHDL 13

Глава 2. Уровни представления и формы абстракции цифровых систем 14

Глава 3. Базовая структура VHDL-файла 18

3.1. Общие сведения 18
3.2. Объявление интерфейса 22
3.3. Архитектура проекта 26
3.3.1. Архитектура проекта в поведенческой форме 26
3.3.2. Архитектура проекта в структурной форме 29
3.4. Библиотеки и пакеты 33

Глава 4. Лексические элементы языка VHDL 37

4.1. Идентификаторы 37
4.2. Ключевые слова 38
4.3. Зарезервированные слова 39
4.4. Числа 40
4.5. Символы, строки и битовые строки 41

Глава 5. Программные элементы данных: константы, переменные и сигналы 42

5.1. Константы 42
5.2. Переменные 43
5.3. Сигналы 44

Глава 6. Типы данных 47

6.1. Встроенные типы данных пакета STANDARD 47
6.2. Пользовательские типы и подтипы 49
6.2.1.Физические типы 50
6.2.2. Перечислимые типы 51
6.2.3. Композитные пользовательские типы 54
6.3. Преобразование типов 61
6.4. Атрибуты 64
6.4.1.Атрибуты сигналов 64
6.4.2.Скалярные атрибуты 65
6.4.3. Атрибуты массивов 68

Глава 7. Операции и символы операций 69

7.1. Логические операции 70
7.2. Операции отношений 70
7.3. Операции сдвига 71
7.4. Операции сложения 72
7.5. Унарные операции 73
7.6. Операции умножения 73
7.7. Вспомогательные операции 74
7.8. Символы комментария 75
7.9. Математические выражения 77

Глава 8. Поведенческая форма проекта: явно заданный оператор Process 79

8.1. Явно заданный оператор PROCESS 79
8.2. Оператор условной передачи управления If 84
8.3. Оператор выбора CASE 86
8.4. Оператор цикла LOOP 89
8.4.1. Базовая форма цикла 89
8.4.2. Итерационная форма цикла WHILE ... LOOP 91
8.4.3. Итерационная форма цикла FOR ... LOOP 92
8.4.4. Операторы NEXT и EXIT 92

8.5. Оператор ожидания WAIT 93
8.6. Оператор NULL 94
8.7. Пример VHDL-проекта 95

Глава 9. Поведенческая форма проекта: неявно заданный оператор PROCESS 97

9.1. Простая параллельная установка значений сигналов 98
9.2. Условная установка значения сигнала 100
9.3. Селективная установка значений сигналов 102

Глава 10. Структурная форма проекта 105

10.1. Оператор COMPONENT 106
10.2. Оператор PORT MAP 107

Глава 11. Примеры VHDL-проектов ЦУ различного назначения 115

11.1. Проекты ЦУ общего назначения 115
11.1.1. Логический элемент AND на 4 входа 115
11.1.2. Логический элемент OR на 2 входа 116
11.1.3. Логический элемент OR на 4 инверсных входа 117
11.1.4. Логический элемент NOR на 3 входа 119
11.1.5. Логический элемент XOR на 2 входа 120
11.1.6. 4-битный буфер — логический усилитель 121
11.1.7. 8-битный буфер с тремя состояниями без обратной связи 122
11.1.8. 4-канальный мультиплексор 4×1 с сигналом разрешения выбора номера канала 122
11.1.9. Демультиплексор 1×4 123
11.1.10. 4-битный последовательный сумматор с фиксированной точкой . . 124
11.1.11. 8-битный сумматор с ускоренным переносом 125
11.1.12. 4-битный каскадный вычитатель с фиксированной точкой 126
11.1.13. 8-битный сумматор со сквозным переносом с учетом знаков битов 127
11.1.14. Универсальный параллельный арифметический процессор 129
11.1.15.4-битный компаратор с анализом знаков сравниваемых чисел и выбором режимов сравнения 130
11.1.16. 8-битный контроллер четности 131
11.1.17. Простой D-триггер, переключаемый передним фронтом тактового импульса 132
11.1.18. D-триггер с асинхронным сбросом 132
11.1.19. D-триггер с асинхронной предустановкой 133
11.1.20. D-триггер с асинхронным сбросом и асинхронной предустановкой 134
11.1.21. D-триггер с синхронным сбросом 134
11.1.22. D-триггер с синхронной предустановкой 135

11.1.23. D-триггер с синхронным сбросом и сигналом, разрешающим приход очередного тактового импульса 136
11.1.24. D-триггер-защелка 137
11.1.25. D-триггер-защелка со стробирующим сигналом 137
11.1.26. 8-битный накапливающий счетчик со счетным входом и асинхронным сбросом 138
11.1.27. 8-битный накапливающий счетчик с синхронной загрузкой и асинхронным сбросом 139
11.1.28. 8-битный накапливающий/вычитающий счетчик с выбором направления счета 140
11.1.29.N-битный накапливающий счетчик со счетным входом, синхронной загрузкой и асинхронным сбросом 141
11.1.30.Синхронный 4-битный циклический сдвиговый регистр вправо с параллельным входом и выходом 142
11.1.31. Универсальный синхронный 4-битный сдвиговый регистр вправо с параллельным входом и выходом 144
11.1.32. Расширитель разрядности шины знаковыми разрядами 145
11.1.33. VHDL-проект расширителя разрядности шины нулями 147
11.1.34. 8-битная двунаправленная шина с тремя состояниями 148
11.1.35. Процессор возведения в степень N 149
11.1.36. Процессор нахождения факториала 149
11.1.37. Процессор нахождения натурального логарифма 150
11.1.38. Процессор нахождения квадратного корня 152
11.1.39. Процессор, конвертирующий 16-ричные значения в данные типа std_logic_vector 153
11.2. Проекты ЦУ специального назначения 154
11.2.1. Генератор тактовых импульсов с периодом следования 100 нс и скважностью 50% 154
11.2.2. Комбинированный генератор псевдослучайных чисел с выходными значениями разных типов 154
11.2.3. Многофункциональный конвертер типов данных 156
11.2.4.Целочисленный мультипликативный криптопроцессор 157
11.2.5. Символьный аддитивный криптопроцессор 158
11.2.6. Подстановочный символьный криптопроцессор 161
11.2.7.Перестановочный битовый криптопроцессор 163
11.2.8.Криптопроцессор на базе сдвиговых операций 166
11.2.9. Криптопроцессор с многоуровневой системой шифрования, формульный метод 168
11.2.10. Криптопроцессор, реализующий метод эллиптических кривых 170
11.3. Проекты ЦУ специального назначения повышенного уровня сложности 177
11.3.1.Файловый криптопроцессор циклического сдвига 177
11.3.2. Файловый RSA-криптопроцессор 184

11.3.3. Файловый криптопроцессор Вижинера. 192

Приложение. Основы языка VHDL в реферативном изложении 199

П.1. Язык VHDL как универсальный язык проектирования ЦУ 199
П.2. Концептуальные положения языка VHDL. 201
П.3. Объявление интерфейса проекта 204
П.4. Программные элементы данных языка VHDL 206
П.5. Поведенческая форма проекта на основе явно заданного оператора PROCESS 210
П.6. Поведенческая форма проекта на основе неявно заданного оператора PROCESS. 212
П.7. Структурная форма проекта. 214

Принятые сокращения

АЛУ — арифметико-логическое устройство.
ОПО — оператор параллельной обработки.
ПО — программное обеспечение.
САПР — система автоматизированного проектирования.
СБИС — сверхбольшая интегральная схема.
ЦУ — цифровое устройство.
ПУЗС — простая установка значения сигнала.
УУЗС — условная установка значения сигнала.
СУЗС — селективная установка значения сигнала.

ГЛАВА 1

ВВЕДЕНИЕ

1.1. Современные языки проектирования цифровых устройств

VHDL является аббревиатурой от Very high speed integrated circuits Hardware Description Language, что переводится как *язык описания устройств на сверхбольших интегральных схемах* (СБИС). В середине 1980-х гг. Министерство обороны США и IEEE[1)] спонсировали разработку этого языка описания цифровой аппаратуры с целью получения простого в использовании средства проектирования и моделирования логических схем для всех этапов разработки электронных систем, начиная от модулей микросхем и кончая крупными вычислительными системами. Первая версия стандарта была издана в 1987 г. (IEEE 1076-1987). Очередные версии выходили в 1991, 1993, 1996, 1997, 1999, 2000 и 2002 гг. В настоящее время действует стандарт VHDL, изложенный в документе IEEE 1076-2002 и являющийся промышленным стандартом, который широко используется для описания работы цифровых систем. В июне 2006 г. была опубликована версия 3.0 проекта стандарта VHDL-2006, в который вошли все дочерние стандарты, разработанные в ходе создания стандартов VHDL (IEEE

[1)] IEEE (Institute of Electrical and Electronic Engineers — Институт инженеров по электротехнике и радиоэлектронике, ИИЭР) — международная организация, созданная в США в 1963 г. Является разработчиком ряда стандартов для локальных вычислительных систем, в том числе по кабельной системе, физической топологии и методам доступа к среде передачи данных. — *Примеч. ред.*

1064, 1076.2, 1076.3), а также добавлены другие усовершенствования, такие как интерфейс с языками высокого уровня C/C++ и ряд других.

В РФ язык VHDL закреплен стандартом ГОСТ РФ 50754-95 «Язык описания аппаратуры цифровых систем VHDL. Описание языка».

В данной книге рассмотрены основные положения языка VHDL, которые определены в стандарте IEEE 1076-1993, поскольку последующие изменения (за исключением пока еще разрабатываемого стандарта VHDL-2006) были несущественными. Кроме того, все последующие стандарты VHDL, включая VHDL-2006, обеспечивают совместимость с проектами, разработанными в соответствии со стандартом 1076-1993.

Еще одним представителем языков описания цифровой аппаратуры является язык Verilog, или Verilog HDL. Разработчики Verilog сделали его синтаксис очень похожим на синтаксис языка C. Verilog имеет препроцессор, очень похожий на препроцессор языка C, а основные управляющие конструкции Verilog также подобны одноименным конструкциям языка C. В настоящее время действует стандарт Verilog, изложенный в документе IEEE 1364-2005, который представляет собой несколько доработанный вариант очень популярного среди приверженцев Verilog стандарта IEEE 1364-2001. Язык Verilog применяется в промышленности так же широко, как и VHDL, поскольку оба этих языка позволяют описывать и имитировать работу сложных цифровых систем.

Третьим представителем языков VHDL является язык ABEL (Advanced Boolean Equation Language — расширенный язык двоичных уравнений), который был разработан в 1983 г. для создания проектов цифровых устройств посредством *программируемых логических устройств* (PLD — Programmable Logic Devices). Язык ABEL менее мощен, чем VHDL и Verilog, в частности не позволяет проектировать устройства на перепрограммируемых логических матрицах FPGA (Field Programmable Gate Arrays), поэтому он менее популярен в промышленности, хотя все еще находит своих сторонников.

1.2. Краткая характеристика языка VHDL

Хотя язык VHDL внешне выглядит так же, как и другие традиционные языки программирования (поскольку имеет литералы, разделители, операторы и т. д.), он обладает некоторыми важными отличительными характеристиками.

- Проекты цифровых устройств (ЦУ), созданные с помощью языка VHDL, имеют, как правило, иерархическую структуру.
- Каждый автономный проектируемый модуль (субблок проектируемого ЦУ) имеет:
  — строго определенный интерфейс взаимодействия с другими модулями;
  — точную спецификацию внутреннего устройства проектируемого модуля, описывающую концепцию и функционирование модуля.
- Спецификации модулей VHDL-проектов могут использовать или математические алгоритмы, описывающие их работу, или описание аппаратной структуры проектируемого модуля. В соответствии с уровнями абстракций проектов описание модуля может иметь *поведенческую* или *структурную* форму.
- Моделирование алгоритма работы проекта основывается на событийном принципе управления.
- VHDL-проект позволяет выполнять моделирование протекания параллельных процессов в электрических схемах, временной анализ сигналов и их параметров.
- VHDL поддерживается инструментальными средствами синтеза и системами автоматизированного проектирования (САПР) многих производителей программного обеспечения (ПО), которые могут создавать прямо из описания VHDL-проекта его аппаратную реализацию (связанные между собой структуры логических элементов, содержащихся в СБИС).
- Используя VHDL, можно проектировать, моделировать и синтезировать практически любое ЦУ, начиная от простой комбинационной схемы до законченной микропроцессорной системы на СБИС.

Перечисленные выше характеристики языка VHDL как специализированного языка описания ЦУ реализуются с помощью следующих языковых средств:

- Библиотеки и пакеты.
- Проекты: интерфейс и архитектура проекта.
- Подпрограммы: функции и процедуры.
- Скалярные типы данных: перечислимые, числовые, физические.
- Программные элементы данных: константы, переменные, сигналы, порты, идентификаторы.

- Математические операции: логические, отношений, арифметические.
- Программные операции: установка значений сигналов, присвоение значений переменным, реализация связи портов и сигналов.
- Математические выражения: логические, алгебраические, логико-алгебраические.
- Операторы объявления программных элементов данных.
- Операторы комбинаторной логики: простой установки значения сигнала (ПУЗС), условной установки значения сигнала (УУЗС), селективной установки значения сигнала (СУЗС), оператор `process`, оператор реализации компонента `port map`.
- Операторы регистровой логики: оператор `process`, ПУЗС, оператор условной передачи управления, оператор цикла, оператор выбора.

В отличие от процедурных языков программирования, языковые средства которых обеспечивают выполнение вычислений над абстрактными данными и управление ими, VHDL-проект описывает ЦУ, учитывая его многогранность, поведение, структуру, функциональные и физические свойства, а также взаимодействие со специальной аппаратурой, физически реализующей проект ЦУ в СБИС.

Остановимся особо на некоторых важных отличиях языка VHDL. VHDL по существу является языком *параллельного программирования*, т. е. в его конструкции существуют операторы, соответствующие логическим вентилям. Эти операторы обрабатываются (т. е. вычисляются) по *параллельному принципу*. Суть данного принципа состоит в том, что, как только сигнал, содержащийся в описании проектируемого ЦУ, изменяет свое значение (говорят, что «происходит событие на сигнале»), все операторы, принимающие участие в его обслуживании, мгновенно запускаются на выполнение и одновременно выдают конечный результат. Поэтому такие операторы называются *операторами параллельной обработки* (ОПО) (concurence operator). Программа, написанная на VHDL (как и на любом другом HDL-языке, например Verilog), моделирует физическое поведение системы (как правило, цифровой), сигналы в которой распространяются мгновенно. Такая программа позволяет формировать временную спецификацию (время задержки распространения сигнала на логическом элементе), а также описывать систему как соединение различного рода компонентов, или функциональных блоков.

1.3. Этапы проектирования с использованием VHDL

В подавляющем большинстве случаев аппаратная реализация проекта ЦУ с использованием VHDL протекает в соответствии со следующими этапами:

Разработка иерархической блок-схемы проекта. Выяснение базового конструктивно-технологического метода и стандартных блоков на уровне структурной схемы. Поскольку большие логические проекты являются, как правило, иерархическими, использование VHDL позволяет легко разбить проект на модули (субпроекты) и определить их интерфейсы.

Программирование. Запись VHDL-кода для модулей и их интерфейсов.

Компиляция. Анализ программного кода VHDL-проекта для выявления синтаксических ошибок, а также проверка его совместимости с другими модулями. В ходе компиляции также собирается внутренняя информация о структуре проекта, которая необходима для моделирования работы проектируемого ЦУ.

Моделирование. Определение и применение входных воздействий к откомпилированному коду проекта с наблюдением выходных реакций. Моделирование может выполняться как в форме *функционального контроля*, т. е. проверки логики работы проекта без учета временных соотношений и задержек распространения сигналов на логических элементах, так и в качестве одного из этапов *верификации* завершенного проекта.

Синтез. Преобразование VHDL-описания в набор примитивов или логических элементов, которые могут быть реализованы с учетом конкретной технологии.

Компоновка, монтаж и разводка. Отображение проекта на карте синтезирующих элементов, содержащихся в СБИС.

Временной анализ. Получение фактических задержек реализованной в СБИС цифровой схемы проекта с учетом длины соединений, электрических нагрузок и других известных факторов.

ГЛАВА 2

УРОВНИ ПРЕДСТАВЛЕНИЯ И ФОРМЫ АБСТРАКЦИИ ЦИФРОВЫХ СИСТЕМ

Цифровая система на языке VHDL может быть представлена на различных уровнях и различными формами абстракции. Представление цифровой системы на нескольких уровнях абстракции показано на **Рис. 2.1**.

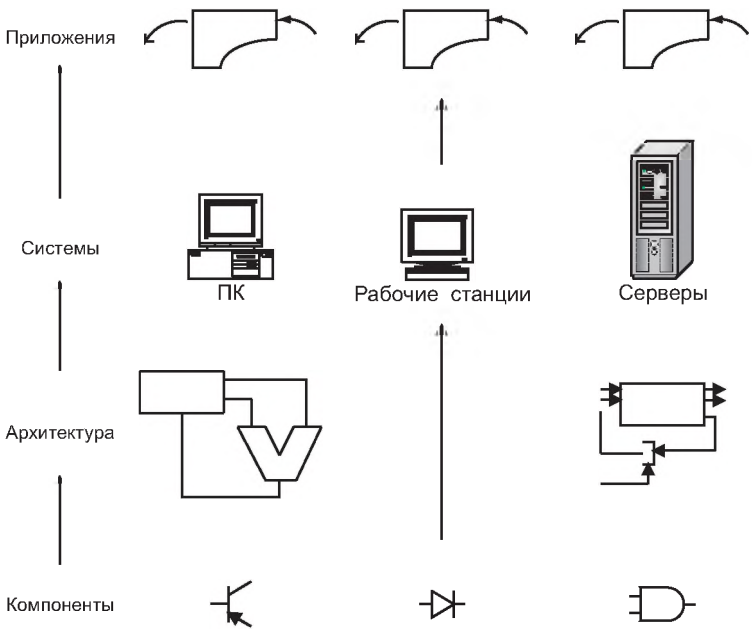

Рис. 2.1. Представление цифровой системы на нескольких уровнях абстракции

Как видно на **Рис. 2.1**, цифровая система может быть описана на уровне компонентов (транзисторов, диодов, логических элементов), на уровне архитектуры (структурной схемы, содержащей АЛУ, регистры, компараторы и т. д.), на уровне автономной системы (ПК, рабочей станции, сервера и т. д.), а также на уровне приложений (программных модулей, входящих в состав систем более высокого уровня).

Различные формы абстракции цифровой системы дают возможность сохранять описание и проект как комплексную управляемую систему. На **Рис. 2.2** показаны различные формы абстракции.

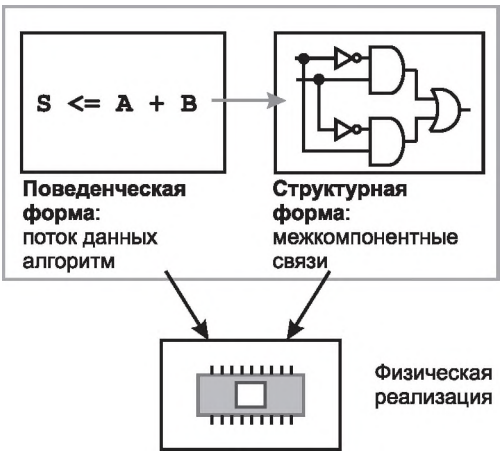

Рис. 2.2. Формы абстракции: поведенческая, структурная и физическая

Первичная и наивысшая форма абстракции — это *поведенческая* (behavioral) форма, которая позволяет описывать систему в терминах того, что она делает (или как она ведет себя), а не в терминах ее элементов либо компонентов и соединений между ними. Поведенческая форма представления определяет зависимость между входными и выходными сигналами. Поведенческая форма может быть булевым (Boolean) выражением либо более абстрактным описанием, например описанием *межрегистровых пересылок* (register transfer) или описанием в *алгоритмической* форме.

В качестве примера представления поведенческой формы рассмотрим простое устройство `SIREN`, которое выдает сигнал предупреждения (`Warning`) всякий раз, когда автомобильный ключ вставлен в замок зажигания (`Ignition_on`), если дверь открыта (`Door_open`) или отстегнут ремень безопасности (`Seatbelt_off`). На поведенческом уровне это словесное описание может быть выражено как:

```
Warning = Ignition_on and (Door_open or Seatbelt_off)
```

Структурная (structural) форма, с другой стороны, представляет систему как набор логических элементов и компонентов, которые связаны между собой таким образом, чтобы выполнить нужную функцию. Структурную форму представления можно сравнить со схемным решением связанных логических вентилей. Структурный уровень является обычно окончательным представлением физической реализации системы. Структурное представление рассмотренного выше примера показано на **Рис. 2.3**.

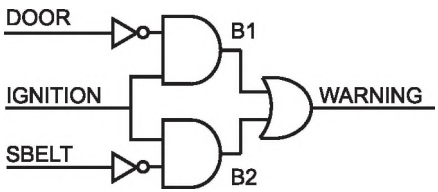

Рис. 2.3. Структурное представление устройства `SIREN`

Как говорилось выше, язык VHDL позволяет описать цифровую систему и в *структурной*, и в *поведенческой* форме. При этом поведенческая форма может быть реализована в одном из двух стилей: *в виде потока данных* (data flow style) и *в алгоритмическом виде* (algoritmic style).

Представление системы в виде потока данных позволяет описать систему с учетом направления потока перемещаемых через нее данных. Этот стиль характерен для описания поведения системы на уровне межрегистровых пересылок.

При представлении системы в алгоритмическом виде поведение системы описывается с помощью *операторов* (operator). При этом для описания поведения системы могут использоваться как операторы *параллельной обработки* (concurrent), которые выполняются параллельно,

как только данные поступают на входы, так и *последовательные* (sequential) операторы, которые выполняются последовательно в порядке их записи. Язык VHDL допускает как параллельную, так и последовательную установку значений сигналов, которые определяют способ и порядок их выполнения. Примеры обоих представлений будут даны в последующих главах книги.

ГЛАВА 3

БАЗОВАЯ СТРУКТУРА VHDL-ФАЙЛА

3.1. Общие сведения

Проект любого ЦУ на языке VHDL — это прежде всего программа, которая содержит *ключевые* и *зарезервированные* слова. Эти слова не могут использоваться как имена сигналов или как идентификаторы. В языке VHDL ключевые слова, зарезервированные слова и определяемые пользователем идентификаторы нечувствительны к регистру.

Строки с комментариями в VHDL-программе начинаются с двух смежных дефисов (--) и при компиляции игнорируются компилятором так же, как символы конца строки и пробелы.

VHDL — это *строго типизированный* язык. Это означает, что все программные элементы данных (константа, переменная, сигнал) должны явно объявляться с обязательным указанием типа элемента. Тип, указанный в объявлении программного элемента данных, определяет информационные характеристики этого элемента, диапазон допустимых числовых значений, которые может принимать этот программный элемент, а также операции, которые могут над ним выполняться. Кроме того, в языке VHDL, в отличие от других строго типизированных языков, *не допускаются* операции над разнотипными элементами без предварительного преобразования типов.

В языке VHDL (как и в классических языках программирования) из ключевых, зарезервированных слов и других лексических элементов строятся образования, которые принято называть *операторами*. Оператором в языке VHDL считается любая запись, начинающаяся с ключевого слова и заканчивающаяся символом точки с запятой (;). Диаграмма, иллюстрирующая одну из возможных классификаций операторов, показана на **Рис. 3.1**.

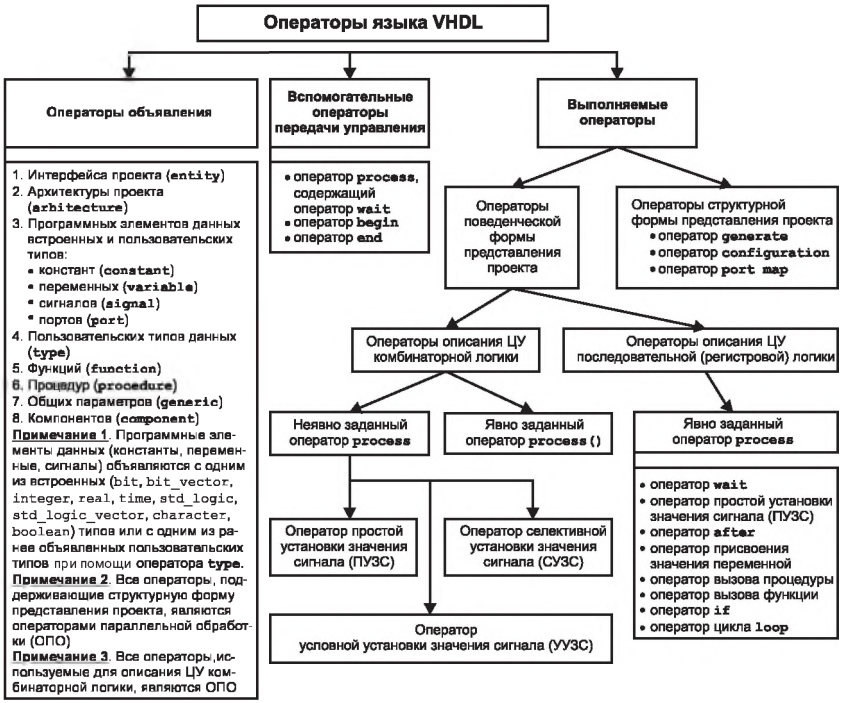

Рис. 3.1. Классификация операторов языка VHDL

В языке VHDL операторы с точки зрения их функционального программного назначения (как при поведенческой, так и при структурной методологии описания проектов) принято делить на две основные группы:

- *Операторы объявления* программных элементов данных и их типов, компонентов проектируемого ЦУ, их интерфейсов и архитектур, пользовательских типов, а также функций и процедур.
- *Выполняемые операторы*.

Кроме основных операторов в языке VHDL принято выделять еще одну группу операторов, которую называют *вспомогательными операторами передачи управления*.

С точки зрения конструктивного устройства операторы обеих групп принято подразделять на два класса:

- **Простые (однострочные) операторы**. Простой оператор представляет собой текстовую запись, располагающуюся, как правило, на одной строке, которая *не содержит* других операторов.
- **Составные (многострочные) операторы.** Составной оператор представляет собой текстовую запись, занимающую, как правило, много строк, которая *содержит* другие операторы.

В структуре языка VHDL имеется два фундаментальных оператора (составные по своему конструктивному устройству), которые поддерживают *принцип системного проектирования*. Принцип системного проектирования предполагает, что любое проектируемое ЦУ рассматривается как *автономная подсистема*, которая должна иметь:

- идентификатор (т. е. должна быть объявлена);
- способность взаимодействовать с другими проектируемыми подсистемами посредством своего интерфейса (входных/выходных портов);
- описание внутренней структуры или алгоритма функционирования.

Такими фундаментальными операторами в языке VHDL являются операторы:

```
entity ... end entity ...;
architecture ... end architecture ...;
```

Проект ЦУ, или его компонент, описанный с помощью языка VHDL, хранится в файле, который обычно снабжается расширением VHD.

Представленный в виде VHD-файла проект ЦУ может содержать другие объекты (подсистемы), которые в таком случае являются подчиненными компонентами системы верхнего уровня.

Любой компонент, независимо от того, является ли он автономным либо подчиненным компонентом, или же системой верхнего уровня, в свою очередь является совокупностью *интерфейса* (**`entity`**) и *архитектуры* (**`architecture`**).

В *объявлении интерфейса* (**`entity`** declaration) содержится *объявление портов* (ports declaration) проектируемого компонента с внешним миром. Объявление портов проектируемого компонента определяет

внешние входные и выходные интерфейсные сигналы, в то время как архитектура представляет собой набор таких взаимосвязанных программных элементов, как подчиненные компоненты, операторы `process`, операторы параллельных вычислений, последовательные операторы, подпрограммы (**Рис. 3.2**). В типичном проекте содержится несколько таких взаимосвязанных объектов, предназначенных для выполнения преобразований, в ходе которых и обеспечивается собственно требуемая функциональность проекта.

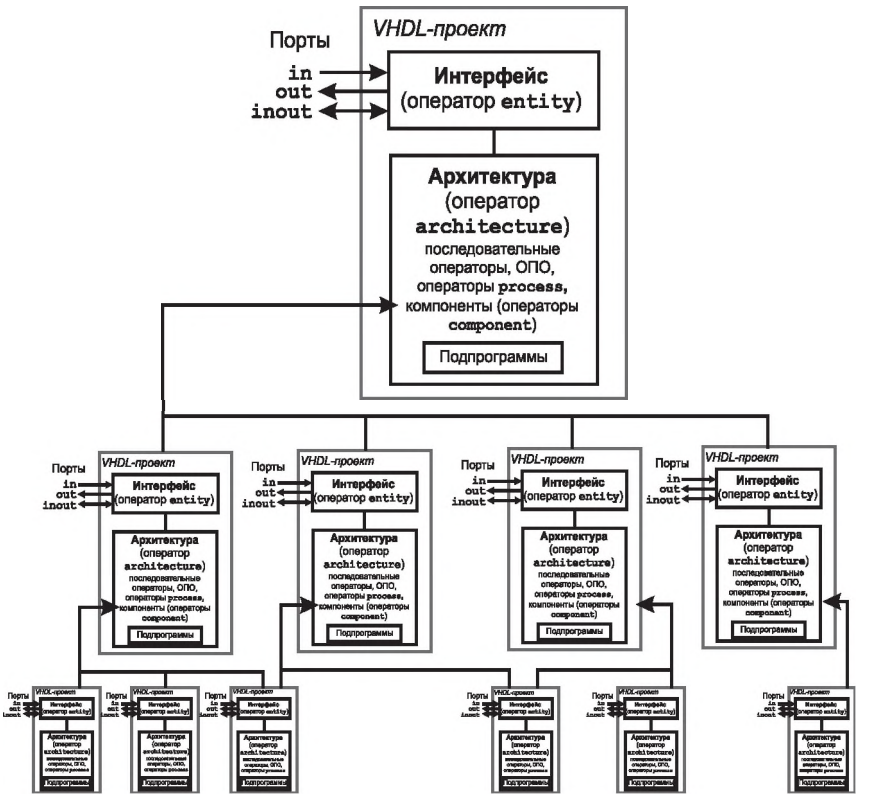

Рис. 3.2. VHDL-проект, представляя собой совокупность интерфейса и архитектуры, может использовать субпроекты в качестве компонентов, а также входить в качестве компонента в VHDL-проекты более высокого уровня иерархии

3.2. Объявление интерфейса

В соответствии с синтаксисом языка VHDL объявление интерфейса проектируемого компонента начинается с ключевого слова **entity** и определяет имя компонента, а также содержит объявление входных и выходных портов. Общий синтаксис объявления интерфейса имеет такой вид:

```
entity ИМЯ_КОМПОНЕНТА is [ generic (общие_объявления);]
   port (сигнал1: режим тип;
         сигнал2: режим тип;
         ...
         сигналN: режим тип);
end [entity] [ИМЯ_КОМПОНЕНТА] ;
```

Как видно из приведенного синтаксиса, объявление интерфейса всегда начинается с ключевого слова **entity**, за которым указываются имя проектируемого компонента и ключевое слово **is**. Далее в объявлении интерфейса с помощью ключевого слова **port** размещается объявление портов. Объявление интерфейса всегда заканчивается ключевым словом **end**, после которого могут указываться ключевое слово **entity**, а также имя компонента (два последних элемента в объявлении интерфейса можно опускать).

Входящие в конструкцию объявления интерфейса проектируемого компонента (т. е. проектируемого ЦУ) элементы имеют следующее назначение:

- *ИМЯ_КОМПОНЕНТА* — определенный пользователем идентификатор проекта.
- *сигнал:* — список разделяемых запятыми идентификаторов, представляющих внешние интерфейсные сигналы (порты проекта).
- *режим* — ключевое слово, показывающее направление действия сигнала (порта):
  - **in** — ключевое слово, используемое для объявления входных сигналов;
  - **out** — ключевое слово, используемое для объявления выходных сигналов. Значения выходных сигналов могут считываться другими проектами;

  - **buffer** — ключевое слово, используемое для объявления внутренних сигналов. Значения сигнала с этого порта могут читаться только внутри архитектуры проекта;
  - **inout** — ключевое слово, используемое для объявления сигналов, которые могут быть как входными, так и выходными.
- *тип* — зарезервированное слово, используемое для объявления определенного пользователем типа сигнала, переменной или константы. Примерами встроенных типов являются: `bit`, `bit_vector`, `Boolean`, `character`, `std_logic` и `stc_ulogic`.
  - `bit` — может принимать значение или 0, или 1;
  - `bit_vector` — зарезервированное слово, показывающее, что сигнал, переменная или константа являются *вектором*, состоящим из значений типа `bit`, например `bit_vector (0 to 7)`;
  - `std_logic`, `std_ulogic`, `std_logic_vector`, `std_ulogic_vector` — зарезервированные слова, показывающие, что сигнал, переменная или константа могут принимать одно из девяти значений, отражающих его силу и реальные физические значения. В проектах ЦУ предпочтительнее использовать типы `std_logic`, `std_ulogic`, чем типы `bit`, `bit_vector`;
  - `boolean` — зарезервированное слово, означающее, что сигнал, переменная или константа могут принимать одно из двух значений: `TRUE` или `FALSE`;
  - `integer` — зарезервированное слово, означающее, что сигнал, переменная или константа могут принимать одно целое значение, попадающее в указанный диапазон;
  - `real` — зарезервированное слово, означающее, что сигнал, переменная или константа могут принимать одно вещественное значение, попадающее в указанный диапазон;
  - `character` — зарезервированное слово, означающее, что сигнал, переменная или константа могут принимать значение печатаемого ASCII-символа;
  - `time` — зарезервированное слово, означающее, что сигнал или константа снабжены единицей измерения времени.

- **generic** (*общие_объявления*)**:** — объявление **generic** является факультативным и определяет локальные константы, используемые для задания временных параметров и разрядности (например, разрядности шины) проектируемого ЦУ. В объявлении **generic** могут указываться значения, используемые по умолчанию. Синтаксис для объявления **generic** следующий:

```
generic (
имя_константы1: тип [:=значение] ;
имя_константы2: тип [:=значение] ;
...
имя_константыN: тип [:=значение]);
```

Для примера ЦУ SIREN, структурное представление которого показано на **Рис. 2.3**, объявление интерфейса имеет следующий вид:

```
-- Пример объявления интерфейса для ЦУ SIREN (Рис. 2.3).
entity SIREN is
   port (DOOR, IGNITION, SBELT: in std_logic;
         WARNING: out std_logic);
end SIREN;
```

Как видно из объявления, проектируемый интерфейс имеет идентификатор SIREN, содержит три входных порта (DOOR, IGNITION и SBELT) и один выходной порт (WARNING). Обратите внимание на порядок их размещения и наличие в оформлении *точек с запятой*! Входные порты обозначены с помощью ключевого слова **in**, а выходные — с помощью ключевого слова **out**. Так как язык VHDL является языком со строгим контролем типов, в объявлении каждого порта имеется указание о его информационном *типе*. Тип определяет набор значений, которые могут присваиваться объекту (в данном случае — сигналу). В этом проекте мы использовали тип std_logic. Это наиболее предпочтительный тип для цифровых сигналов. В отличие от типа bit, который обеспечивает поддержку только бинарных значений '1' или '0', типы std_logic и std_ulogic позволяют присваивать соответствующим элементам данных одно из девяти значений.

Использование типа std_logic и std_ulogic дает возможность точнее описать значения сигналов физической цифровой системы. В

общем случае в физической цифровой системе сигнал может принимать бинарные значения '0' или '1', 'X' (неизвестное значение), 'U' (неициализированное значение), '-' (безразличное состояние) и ряд других (полный перечень значений приведен в подразд. 6.2.2). Определение типа std_logic находится в пакете std_logic_1164 библиотеки ieee. Тип определяет набор значений, которые объект может иметь.

Приведем еще два примера объявлений интерфейсов проектируемых объектов.

Объявление интерфейса 8-битного мультиплексора 4×1

```
entity mux4_to_1 is
   port (I0,I1,I2,I3: in std_logic_vector(7 downto 0);
         OUT1: out std_logic_vector(7 downto 0));
end mux4_to_1;
```

Объявление интерфейса D-триггера со входами установки и сброса

```
entity dff_sr is
   port (D,CLK,S,R: in std_logic;
         Q,Qnot: out std_logic);
end dff_sr;
```

Примечание. *В языке VHDL любой конструктивный элемент, начинающийся некоторым ключевым словом и заканчивающийся ключевым словом* ***end*** *и символом «;» (точка с запятой), принято называть составным оператором. Однако операторы могут быть не только составными, т. е. содержащими в себе другие операторы (часто говорят, что составные операторы «содержат тело»), но и простыми. Простые операторы размещаются обычно в одной строке, не имеют ключевого слова* ***end*** *и всегда заканчиваются символом «;» (точка с запятой).*

Анализируя синтаксис объявления интерфейса, можно заключить следующее: любой проект ЦУ реализуется в языке VHDL с помощью оператора **entity.. end** [**entity**]**...;**. Этот оператор по своей конструкции является составным оператором и содержит другие операторы. Так, в приведенных выше примерах составной оператор имеет в своей конструкции оператор **port(...);**.

3.3. Архитектура проекта

Архитектура проекта (architecture body) определяет логику работы схемы проектируемого ЦУ, т. е. устанавливает алгоритм работы ЦУ, и форму ее реализации (поведенческая или структурная).

Примечание. *Как говорилось выше, VHDL-проект ЦУ может иметь разные формы представления: поведенческую, структурную или их комбинацию.*

Синтаксис объявления архитектуры проекта имеет следующий вид:

```
architecture Имя_архитектуры of ИМЯ_КОМПОНЕНТА is
        -- Объявления:
        -- компонентов
        -- сигналов
        -- констант
        -- функций
        -- процедур
        -- типов
...
begin
        -- Выполняемые операторы параллельной обработки.
...
end [architecture] Имя_архитектуры;
```

3.3.1. Архитектура проекта в поведенческой форме

Поведенческая форма архитектуры проекта, представленного на **Рис. 2.3**, имеет следующий вид:

```
architecture behav of SIREN is
begin
WARNING <= (not DOOR and IGNITION) or (not SBELT and IGNITION);
end behav;
```

В строке заголовка архитектуры проекта присутствует идентификатор архитектуры (в данном примере — `behav`), который играет роль имени архитектуры и связывает эту архитектуру с объявлением интерфейса `SIREN`. В качестве имени архитектуры может использоваться любой идентификатор, выбираемый пользователем, при условии соот-

ветствия его синтаксиса требованиям языка VHDL. В рассматриваемом примере компонент SIREN не имеет явно оформленного раздела объявлений вследствие его ненадобности. Основная часть архитектуры проекта (раздел выполняемых операторов параллельной обработки) начинается ключевым словом **begin**. Он обычно содержит описание внутреннего устройства (т. е. функциональности) проекта в форме булевых функций, оформленных в виде операторов параллельной обработки языка VHDL.

Как говорилось выше, поведенческая форма проекта может быть описана несколькими способами. В частности, в этом примере основная часть архитектуры проекта содержит один простой оператор установки значения сигнала. В данном операторе имеется символ «<=», который означает, что сигналу, стоящему слева от символа «<=», устанавливается результат вычисления выражения, стоящего справа от символа «<=». Основная часть архитектуры проекта заканчивается ключевым словом **end**, которое может дополняться ключевым словом **architecture** и именем архитектуры.

Рассмотрим еще два примера оформления архитектуры проекта.

Интерфейс и архитектура (поведенческая форма) двухвходового логического элемента AND

```
entity AND2 is
   port (in1, in2: in std_logic;
         out1: out std_logic);
end AND2;
architecture behavioral_2 of AND2 is
begin
out1 <= in1 and in2;
end behavioral_2;
```

Интерфейс и архитектура (поведенческая форма) двухвходового логического элемента XNOR

```
entity XNOR2 is
   port (A, B: in std_logic;
             Z: out std_logic);
end XNOR2;
architecture behavioral_xnor of XNOR2 is
   -- Объявление внутренних сигналов X и Y.
```

```
   signal X, Y: std_logic;
begin
X <= A and B;
Y <= (not A) and (not B);
Z <= X or Y;
end behavioral_xnor;
```

Операторы установки значения сигнала («<=») в этих примерах содержат логические операции **and**, **or** и **not**. Кроме указанных, в таких операторах могут использоваться операции **nand**, **nor**, **xor** и **xnor**.

Примечание. *Следует особо заметить, что операторы установки значения сигнала (signal assignment statement) в вышеупомянутых примерах являются операторами параллельной обработки (concurrent processing). Такие операторы выполняются параллельно, когда один или больше сигналов, входящих в состав выражений, которые расположены в правой части оператора, изменяют свое значение (т. е. когда хотя бы на одном из сигналов происходит событие). Например, когда сигнал порта* `A` *изменит свое значение, одновременно изменят свои значения внутренние сигналы* `X` *и* `Y`*, которые в свою очередь одновременно (во всяком случае в идеале) заставят последний оператор модифицировать значение сигнала на выходном порту* `Z`*. (В реальных цифровых системах при распространении сигналов может существовать задержка, связанная с этими изменениями.)*

Физические ЦУ управляются данными, а также событиями, возникающими при изменении данных. То есть событие, которое происходит на одном из сигналов, приводит к появлению события на другом сигнале, и т. д. Выполнение операторов детерминировано потоком изменения значений сигналов, т. е. определено последовательностью возникновения событий на сигналах. В результате такого принципа функционирования порядок, в котором операторы параллельной обработки располагаются в основной части архитектуры проекта, может быть любым. Иными словам, если разместить оператор `Z` *перед операторами* `X` *и* `Y`*, конечный результат от этого не изменится. В этом состоит главное отличие языка VHDL от языков программирования, в которых операторы выполняются последовательно или путем вызова процедур.*

3.3.2. Архитектура проекта в структурной форме

Проект, показанный на **Рис. 2.3**, можно также создать, используя структурную форму архитектуры, которая сводится к описанию проекта через набор логических элементов, или компонентов, и связей между их входами и выходами. Вариант такого структурного описания имеет следующий вид:

```
architecture struct of SIREN is
       -- Объявление компонентов.
component AND2
             port (in1, in2: in std_logic;
                   out 1: out std_logic);
end component;
component OR2
             port (in1, in2: in std_logic;
                   out 1: out std_logic);
end component;
component NOT1
             port (in1: in std_logic;
                   out 1: out std_logic);
end component;
-- Объявление сигналов, используемых для связей компонентов.
signal DOOR_NOT, SBELT_NOT, B1, B2: std_logic;
begin
   -- Операторы реализации компонентов.
   U0: NOT1 port map (DOOR, DOOR_NOT);
   U1: NOT1 port map (SBELT, SBELT_NOT);
   U2: AND2 port map (IGNITION, DOOR_NOT, B1);
   U3: AND2 port map (IGNITION, SBELT_NOT, B2);
   U4: OR2 port map (B1, B2, WARNING);
end struct;
```

Если архитектура проекта имеет структурную форму, после заголовка архитектуры располагаются объявления компонентов, которые предполагается использовать в проекте. В приведенном выше примере проекта SIREN использованы два двухвходовых логических элемента AND, один двухвходовой логический элемент OR и инвертор. Эти компоненты должны быть определены, т. е. их VHDL-проекты должны уже существовать (объявления интерфейсов этих проектов приведены в предыдущих примерах). Все VHDL-проекты компонентов ЦУ также

могут быть сохранены в виде одного *пакета* (package), доступ к которому можно получить включая объявления библиотек и пакетов в заголовок проекта (*см.* далее разд. 3.4). Объявления компонентов содержат описания входных (например, `in1`, `in2`) и выходных (например, **`out`** `1`) портов.

Затем необходимо определить внутренние связи путем объявления именованных сигналов. В нашем примере эти внутренние сигналы имеют имена `DOOR_NOT`, `SBELT_NOT`, `B1`, `B2` (*см.* **Рис. 2.3**). Обратите внимание на то, что в объявлении сигнала всегда требуется указывать его тип.

При структурной форме в разделе выполняемых операторов архитектуры проекта (т. е. после ключевого слова **`begin`**) должны располагаться операторы реализации компонентов, а также операторы, описывающие взаимосвязь компонентов между собой. Каждый оператор реализации создает новый уровень иерархии и начинается с имени реализуемого экземпляра компонента (например, `U0`), сопровождаемого двоеточием, идентификатором самого компонента и ключевым словом **`port map`**. Говорят, что оператор **`port map`** реализует междукомпонентные связи объявляемого в нем конкретного экземпляра компонента.

В частности, в рассматриваемом примере междукомпонентные связи экземпляра `U0` компонента `NOT1` определены посредством *позиционной* (неявной) реализации: внешний сигнал `DOOR` соответствует входу `in1` логического компонента `NOT1`, а внутренний сигнал `DOOR_NOT` — выходу компонента `NOT1`. Точно так же для экземпляра `U2: AND2` первые два сигнала (`IGNITION` и `DOOR_NOT`) соответствуют входам `in1` и `in2` компонента `AND2` соответственно, а сигнал `B1` — выходу **`out`** `1` этого компонента.

Альтернативный метод определения межкомпонентных связей состоит в использовании *именованной* (явной) релизации связей между портами. В этом случае синтаксис оператора реализации имеет следующий вид:

```
экземпляр: компонент port map (порт1 => сигнал1, порт2 => сигнал2,
... портN => сигналN);
U0: NOT1 port map (in1 => DOOR, out 1 => DOOR_NOT);
U1: NOT1 port map (in1 => SBELT, out 1 => SBELT_NOT);
U2: AND2 port map (in1 => IGNITION, in2 => DOOR_NOT, out 1 => B1);
U3: AND2 port map (in1 => IGNITION, in2 => SBELT_NOT, B2);
U4: OR2 port map (in1 => B1, in2 => B2, out 1 => WARNING);
```

Обратите внимание на то, что порядок, в котором эти операторы приведены в разделе выполняемых операторов архитектуры проекта, не имеет никакого значения и не сказывается на конечном результате, так как эти операторы выполняются параллельно. Это объясняется тем, что схемное решение, которое описано этими операторами (*см.* **Рис. 2.3**), не зависит от порядка расположения операторов, поскольку оно фактически определяется списком выбранных компонентов и жестко установленными соединениями между ними.

Теперь вернемся к тому обстоятельству, что структурная форма архитектуры, как уже отмечалось выше, позволяет создать иерархический VHDL-проект, в котором заранее определенные компоненты можно использовать многократно. То же самое относится и к самому высокоуровневому проекту, который в свою очередь может являться компонентом для проекта еще более высокого уровня иерархии, и т. д. Это свойство VHDL-проектов позволяет значительно упростить графические изображения больших интегральных ЦУ. Таким образом, несложно прийти к заключению о том, что иерархическое проектирование всегда предпочтительнее, чем одноуровневое.

Давайте рассмотрим использование иерархической технологии проектирования на примере проекта 4-битного сумматора, показанного на **Рис. 3.3**. Как известно, любой полный однобитный сумматор может быть описан следующими булевыми выражениями, используемыми для получения суммы (`sum`) и сигнала переноса (`carry`).

$$sum = (A \oplus B) \oplus C;$$

$$carry = AB + C(A \oplus B).$$

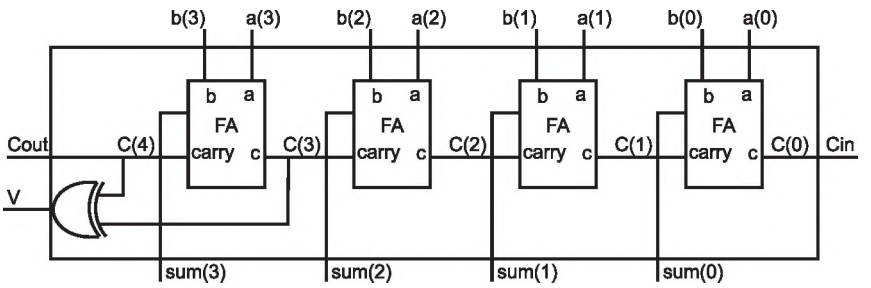

Рис. 3.3. Схема 4-битного сумматора, построенная на полных однобитных сумматорах

В VHDL-проекте FOURBITADD мы определим субпроект (т. е. VHDL-проект более низкого уровня иерархии) полного однобитного сумматора FULLADDER, а затем используем его в качестве компонента основного проекта. Обратите внимание на то, что архитектура fulladder_behav проекта FULLADDER имеет компонентную форму, поскольку в ней не нужно использовать несколько экземпляров одного и того же компонента. Архитектурой fouradder_structure проекта FOURBITADD, имеющей структурную форму, реализуется четыре экземпляра полного однобитного сумматора. Обратите внимание также на ключевое слово **use**, задействованное в проекте FOURBITADD, с помощью которого к проекту подключаются внешние библиотеки и пакеты (подробнее о них рассказывается далее).

```
-- Пример проекта 4-битного сумматора.
library ieee;
use ieee.std_logic_1164.all;
-- Субпроект полного однобитного сумматора.
-- Объявление интерфейса FULLADDER.
entity FULLADDER is
   port (a, b, c: in std_logic;
         sum, carry: out std_logic);
end FULLADDER;
-- Архитектура проекта FULLADDER.
architecture fulladder_behav of FULLADDER is
begin
sum <= (a xor b) xor c ;
carry <= (a and b) or (c and (a xor b));
end fulladder_behav;

-- Основной проект 4-битного сумматора.
library ieee;
use ieee.std_logic_1164.all;
-- Объявление интерфейса FOURBITADD.
entity FOURBITADD is
   port (a, b: in std_logic_vector(3 downto 0);
          Cin: in std_logic;
          sum: out std_logic_vector (3 downto 0);
      Cout, V: out std_logic);
end FOURBITADD;
-- Архитектура проекта FOURBITADD.
```

```
architecture fouradder_structure of FOURBITADD is
   signal c: std_logic_vector (4 downto 0);
component FULLADDER
      port(a, b, c: in std_logic;
        sum, carry: out std_logic);
end component;
begin
      FA0: FULLADDER
           port map (a(0), b(0), Cin, sum(0), c(1));
      FA1: FULLADDER
           port map (a(1), b(1), C(1), sum(1), c(2));
      FA2: FULLADDER
           port map (a(2), b(2), C(2), sum(2), c(3));
      FA3: FULLADDER
           port map (a(3), b(3), C(3), sum(3), c(4));
      V <= c(3) xor c(4);
      Cout <= c(4);
end fouradder_structure;
```

Обратите внимание на то, что в проекте 4-битного сумматора использованы те же самые имена для входных портов, что и для портов полного однобитного сумматора (они названы `a` и `b`). Это не вызывает проблем в языке VHDL, так как эти названия относятся к различным уровням иерархического проекта. Однако для облегчения восприятия текстов VHDL-проектов на практике желательно назначать более осмысленные и уникальные имена.

3.4. Библиотеки и пакеты

Библиотека (library) в VHDL — это нечто вроде каталога, в котором компилятор получает недостающую информацию, используемую в разрабатываемом проекте, но отсутствующую в явном виде в самом проекте. В качестве *пакета* (package) компиляторы VHDL используют библиотечные файлы или модули, которые содержат объявления наиболее задействованных объектов: типов данных, компонентов, сигналов, процедур и функций и т. п. Применение пакетов позволяет подобные объекты многократно использовать в разных VHDL-проектах.

Например, как отмечалось выше, тип данных **`std_logic`** определен в пакете `ieee.std_logic_1164`, который входит в библиотеку

ieee. Поэтому, чтобы в VHDL-проекте использовать тип **std_logic**, требуется подключить к проекту соответствующие библиотеку и пакет. Эти операции выполняются в начале VHDL-проекта с помощью ключевых слов **library** и **use** соответственно, например, таким образом:

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

Расширение .all означает, что в проекте можно использовать описания всех объектов, содержащихся в пакете ieee.std_logic_1164.

Большинство САПР VHDL поддерживают работу с библиотекой ieee, в состав которой входят следующие пакеты:

- **std_logic_1164**. Содержит определения основных типов данных.
- **std_logic_arith**. Содержит арифметические функции, функции преобразования и сравнения, работающие с такими типами данных, как signed, unsigned, integer, std_ulogic, std_logic, std_logic_unsigned и std_logic_vector.
- **std_logic_misc**. Содержит определения вспомогательных типов данных, а также подтипов, констант и функций, используемых пакетом **std_logic_1164**.

Чтобы задействовать любой из этих пакетов, нужно включить в состав проекта операторы объявления требуемых библиотек и пакетов вида

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

Очень популярной библиотекой также является библиотека SYNOPSYS, которая содержит пакет SYNOPSYS.attributes.all с описаниями атрибутов:

```
library SYNOPSYS;
use SYNOPSYS.attributes.all;
```

Разработчик ЦУ может создать и затем подключить к проекту собственный пакет. Объявление пакета должно удовлетворять следующему синтаксису:

```
-- Интерфейс пакета.
package имя_пакета is
   объявления_пакета
end package имя_пакета;
-- Тело пакета.
package body имя_пакета is
   операторы_объявлений_пакета
end package body имя_пакета;
```

Например, базовые логические функции AND2, OR2, NAND2, NOR2, XOR2 и т. д. необходимо предварительно определить как компоненты, прежде чем их можно будет использовать в этом качестве в разрабатываемом проекте. Определить базовые логические функции в качестве компонентов можно, например, в пакете с именем basic_func таким образом:

```
-- Интерфейс пакета.
library ieee;
use ieee.std_logic_1164.all;
package basic_func is
-- объявление AND2.
component AND2
   generic (DELAY: time :=5 ns);
   port (in1, in2: in std_logic; out 1: out std_logic);
end component;
-- объявление OR2.
component OR2
   generic (DELAY: time :=5 ns);
   port (in1, in2: in std_logic; out 1: out std_logic);
end component;
end package basic_func;
-- Тело пакета.
library ieee;
use ieee.std_logic_1164.all;
package body basic_func is
-- 2-входовый вентиль and
entity AND2 is
   generic (DELAY: time);
   port (in1, in2: in std_logic; out 1: out std_logic);
end AND2;
architecture model_conc of AND2 is
```

```
begin
   out1 <= in1 and in2 after DELAY;
end model_conc;
-- 2-входовый вентиль OR
entity OR2 is
   generic (DELAY: time);
   port (in1, in2: in std_logic; out 1: out std_logic);
end OR2;
architecture model_conc2 of AND2 is
begin
   out1 <= in1 or in2 after DELAY;
end model_conc2;
end package body basic_func;
```

Обратите внимание на то, что в интерфейсе пакета в подразделе **component** указано время задержки 5 нс (DELAY: time :=5 ns). Следует заметить, что явное указание времени задержки игнорируется многими инструментальными средствами синтеза. В конструкции разрабатываемого пакета использован встроенный тип **std_logic**, который объявлен в пакете std_logic_1164. Для достижения видимости описания типа **std_logic** из разрабатываемого пакета в него включены операторы **library** ieee; и **use** ieee.std_logic_1164.all;.

Созданный пакет должен быть откомпилирован и помещен в пользовательскую библиотеку. Такую библиотеку можно именовать как my_func. Чтобы использовать компоненты из пользовательской библиотеки my_func, в создаваемом проекте требуется подключить эту библиотеку my_func и пакет basic_func с помощью операторов следующего вида:

```
library ieee, my_func;
use ieee.std_logic_1164.all, my_func.basic_func.all;
```

Внимание! *Язык VHDL допускает перечисление имен библиотек и пакетов после ключевых слов* **library** *и* **use** *соответственно, разделяя их запятыми. Естественно, что можно объявлять в создаваемом проекте библиотеки и пакеты с помощью отдельных операторов, как показано для примера 4-битного сумматора.*

ГЛАВА 4

ЛЕКСИЧЕСКИЕ ЭЛЕМЕНТЫ ЯЗЫКА VHDL

4.1. Идентификаторы

Идентификаторы (identifiers) — это определяемые пользователем слова, используемые для обозначения программных объектов (проектов, интерфейсов, архитектур, пакетов, сигналов, переменных, констант, компонентов и др.) в VHDL-проектах. Выше уже приводились примеры идентификаторов для входных и выходных сигналов, а также для объявлений интерфейсов и архитектур проектов. При выборе идентификаторов следует соблюдать следующие правила:

- Идентификатор может содержать прописные и (или) строчные символы латинского алфавита и цифры (`A...Z`, `a...z`, `0...9`), разделяемые *одним* символом подчеркивания.
- Первым символом в идентификаторе должен быть символ латинского алфавита. Символ подчеркивания не может быть первым или последним символом идентификатора.
- Использовать подряд два символа подчеркивания в идентификаторе запрещается.
- Строчные и прописные символы латинского алфавита в идентификаторе считаются одинаковыми, т. е. идентификаторы `And2`, `AND2` или `and2` равнозначны. Другими словами, идентификаторы не чувствительны к регистру символов.
- Идентификатор может иметь любую длину.

Примеры корректных идентификаторов: `X10`, `x_10`, `My_gate1`.

Примеры некорректных идентификаторов: `_X10`, `my_gate@input`, `gate-input`.

Идентификаторы, создаваемые по приведенным выше правилам, называют *базовыми идентификаторами* (baze identifier). Однако эти правила часто слишком ограничивать разработчиков, не позволяя создавать имена сигналов с особыми свойствами. Например, если в VHDL-проекте требуется обозначить сигнал RESET с активным низким уровнем, проще всего использовать идентификатор `\RESET`. Ограничения, наложенные на базовые идентификаторы, сняты в группе так называемых *расширенных идентификаторов* (extended identifier). К расширенным идентификаторам применяются все те же правила, что и для базовых, но с учетом следующих особенностей:

- Расширенные идентификаторы могут начинаться и заканчиваться символом обратной косой черты «\».
- Расширенные идентификаторы должны отличаться от *зарезервированных слов, ключевых слов* и любых других базовых идентификаторов. Например можно использовать расширенный идентификатор `\if`, который не будет восприниматься компилятором в качестве ключевого слова **`if`**.
- Внутри расширенного идентификатора также может содержаться символ обратной косой черты. В такой ситуации он должен предваряться дополнительным символом обратной косой черты, т. е. данных символов внутри расширенного идентификатора должно быть два.

Расширенные идентификаторы поддерживаются только в версии языка VHDL-93. В версии VHDL-87 они отсутствуют.

Примеры корректного оформления расширенных идентификаторов: `Input`, `\Input\`, `\input#1\`, `\Rst\\as\`.

4.2. Ключевые слова

Некоторые идентификаторы используются системой проектирования как *ключевые слова* (keyword) для организации особых, специфических конструкций языка (операторов языка). Эти ключевые слова не могут применяться как идентификаторы для сигналов или объектов, которые мы определяем в создаваемом проекте. Некоторые из этих ключевых слов приводились ранее: **`in`**, **`out`**, **`or`**, **`and`**, **`port`**, **`map`**, **`end`** и т. д. Ключевые слова часто выделяются САПР VHDL полужирным шрифтом или цветом. Полный список ключевых слов языка VHDL:

```
abs             exit            not             signal
access          file            null            shared
after           for             of              sla
alias           function        on              sll
all             generate        open            sra
and             generic         or              srl
architecture    group           others          subtype
array           guarded         out             then
assert          if              package         to
attribute       impure          port            transport
begin           in              postponed       type
block           inertial        procedure       unaffected
body            inout           process         units
buffer          is              pure            until
bus             label           range           use
case            library         record          variable
component       linkage         register        wait
configuration   literal         reject          when
constant        loop            rem             while
disconnect      map             report          with
downto          mod             return          xnor
else            nand            rol             xor
elsif           new             ror
end             next            select
entity          nor             severity
```

4.3. Зарезервированные слова

Некоторые идентификаторы являются в конкретных САПР VHDL *зарезервированными*. Зарезервированные идентификаторы используются для указания информационных характеристик (типов) программных объектов в операторах их объявлений. В частности, зарезервированными идентификаторами являются такие идентификаторы, как **bit**, **bit_vector**, **std_logic**, **std_logic_vector** и т. д. Зарезервированные идентификаторы *запрещено* применять в качестве базовых идентификаторов.

Стандарт VHDL-87 поддерживает следующие зарезервированные слова: **bit**, **bit_vector**, **integer**, **float**, **string**, **character**.

Стандарт VHDL-93 дополняет этот перечень следующими зарезервированными идентификаторами: **std_logic**, **std_logic_vector**, **std_logic_signed**, **std_logic_unsigned**.

4.4. Числа

По умолчанию *числа*, или числовые литералы, в языке VHDL представляются в десятичной системе счисления. Язык VHDL поддерживает работу с целочисленными (**integer**) и действительными (**real**) литералами. Целочисленный литерал состоит из целого числа без десятичной точки, а действительный литерал всегда содержит десятичную точку. Экспоненциальное представление как целых, так и действительных чисел поддерживается путем использования символов «**E**» или «**e**». Для целочисленных литералов экспонента должна всегда быть положительным числом.

Примеры целочисленных литералов: `12  10  256E3  12e+6`

Примеры вещественных литералов: `1.2  256.24  3.14E-2`

Число `-12` является комбинацией операции минус и целочисленного литерала.

Язык VHDL также поддерживает работу с числами других систем счисления. Такие числа принято называть *базированными* литералами. Синтаксис представления базированных литералов имеет вид

база`#`*число*`#`.

Ниже приведено несколько примеров представления десятичных чисел `18` и `29` с помощью базированных литералов для разных оснований систем счисления.

- Двоичная система: `2#10010#` (эквивалент десятичного числа `18`); `2#11101#` (эквивалент десятичного числа `29`).
- Восьмеричная система: `8#22` (десятичное `18`); `8#35` (десятичное `29`).
- Шестнадцатеричная система: `16#12#` (десятичное `18`); `16#1D#` (десятичное `29`).

Для повышения наглядности при записи больших чисел допускается вставлять символы подчеркивания между группами цифр (например, тетрадами) базированного литерала, например `2#1001_1101_1100_0010#`, `8#215_123#` и т. п.

4.5. Символы, строки и битовые строки

Отдельные символьные литералы, используемые в коде VHDL-проекта, должны заключаться в одинарные прямые кавычки, например:

```
'a', 'B', ','
```

Строки символов заключаются в двойные кавычки, например:

```
"Пример строки."
"Для использования двойных кавычек внутри строки нужно удваивать
соответствующий символ."
"Пример ""вложенной"" строки."
```

В состав строки может быть включен любой *печатаемый* символ.

Битовая строка (bit-string) представляет собой последовательность значений, представленных в виде набора битов. Чтобы указать, что строка является не обычным текстом, а битовой строкой, последняя должна предваряться символом, который обозначает основание системы счисления. В частности, для обозначения двоичной системы счисления используется символ `B`, восьмеричной — символ `O`, а шестнадцатеричной — символ `X`. Например: `B"1100_1001"`, `b"1001011"`, `O"311"`, `o"113"`, `X"C9"`, `x"4b"`.

Внимание! *Следует заметить, что при записи битовых строк в шестнадцатеричной системе счисления каждая цифра соответствует одной 4-битной тетради. Поэтому битовая строка* `b"1001011"`*, представляющая 7 бит, не равна битовой строке* `x"4b"`*, поскольку последняя соответствует 8 битам. По этой же причине битовая строка* `o"113"` *(9 бит) не равна битовой строке* `x"4b"` *(8 бит).*

ГЛАВА 5

ПРОГРАММНЫЕ ЭЛЕМЕНТЫ ДАННЫХ: КОНСТАНТЫ, ПЕРЕМЕННЫЕ И СИГНАЛЫ

Любой программный элемент данных в языке VHDL, прежде чем использоваться в проекте, должен быть *объявлен* (declared) с указанием *типа* (type) и *значения* (value). Программным элементом данных может быть *константа* (constant), *переменная* (variable), *сигнал* (signal), а также *файл* (file). До сих пор мы имели дело с сигналами, которые использовали как входные или выходные порты или в качестве внутренних связей. Сигналы можно рассматривать как программное представление проводников (wire) в схемном решении. Сигналы могут иметь текущее значение и определенный набор возможных значений, являясь функциями операторов *установки значения сигнала* (signal assignment). Переменные и константы используются в качестве вспомогательных программных элементов, используемых для моделирования поведения схем, в операторах **process**, подпрограммах-процедурах и подпрограммах-функциях, подобно тому как они используются в других языках программирования. Давайте кратко обсудим назначение и синтаксис каждого класса этих программных элементов данных.

5.1. Константы

Константа — это программный элемент данных, значение которого остается всегда *постоянным* в процессе моделирования и работы проекта. Объявление программной величины константы имеет следующий синтаксис:

```
constant имя_константы: тип [ := начальное_значение] ;
```

В операторе объявления константы может отсутствовать та часть объявления, в которой указывается начальное значение константы, т. е. [:= *начальное_значение*]. Константы могут быть объявлены в разделе объявлений архитектуры проекта. В этом случае константа является видимой в пределах всей архитектуры того проекта, в котором она объявлена. Константы, объявленные в разделе объявлений оператора **process**, видимы только в пределах этого оператора **process** и могут использоваться только в этом конкретном операторе. Константы могут также объявляться в пакетах. Если константы имеют одинаковый тип и одинаковое начальное значение, их идентификаторы можно перечислять в операторе объявления в виде списка, отделяя одно имя от другого запятой.

Ниже приведены примеры операторов объявлений констант.

```
constant RISE_FALL_TME: time := 2 ns;
constant DELAY1: time := 4 ns;
constant RISE_TIME, FALL_TIME: time:= 1 ns;
constant DATA_BUS: integer:= 16;
```

5.2. Переменные

Переменная — это программный элемент данных, который в процессе моделирования работы проекта может принимать любое, но всегда *единственное* значение в определенный момент. В другой момент переменная может иметь другое значение. Значение переменной может быть модифицировано оператором присваивания значения переменной. При этом считается, что значение переменной модифицируется оператором присваивания значения *немедленно* (т. е. без временной задержки), как только этот оператор выполнен. Переменные должны объявляться только в подразделе объявлений оператора **process**. Объявление переменной имеет следующий синтаксис:

```
variable имя_переменной: тип [ := начальное_значение];
```

Ниже приведены примеры операторов объявлений переменных.

```
variable CNTR_BIT: bit :=0;
variable VAR1: boolean :=FALSE;
variable SUM: integer range 0 to 256 :=16;
variable STS_BIT: bit_vector (7 downto 0);
```

В рассмотренном примере переменная SUM является целым числом, которое имеет диапазон от 0 до 256 и начальное значение 16 в начале процесса моделирования. В четвертом примере определен *битовый вектор* STS_BIT, содержащий 8 элементов, к каждому из которых можно обращаться по имени переменной с указанием *индекса* бита в векторе: STS_BIT (7), STS_BIT (6), … STS_BIT (0).

Как говорилось выше, значение переменной может модифицироваться с помощью оператора присвоения значения переменной. Этот оператор имеет следующий синтаксис:

```
Имя_переменной := выражение;
```

Если переменные имеют одинаковый тип и одинаковое начальное значение, их идентификаторы можно перечислять в операторе объявления в виде списка, отделяя одно имя от другого запятой.

5.3. Сигналы

Сигнал — это основной программный элемент данных в VHDL-проектах, предназначенный для переноса информации между компонентами. Фактически сигналы связывают компоненты проекта в единую систему. Оператор объявления сигнала имеет следующий синтаксис:

```
signal имя_сигнала: тип [ := начальное_значение];
```

Если сигналы имеют одинаковый тип и одинаковое начальное значение, их идентификаторы можно перечислять в операторе объявления в виде списка, отделяя одно имя от другого запятой.

Ниже приведены примеры операторов объявлений сигналов.

```
signal SUM, CARRY: std_logic;
signal CLOCK: bit;
signal TRIGGER: integer :=0;
signal DATA_BUS: bit_vector (0 to 7);
signal VALUE: integer range 0 to 100;
```

Значения сигналов изменяются при выполнении операторов установки значения сигналов. В отличие от переменных в операторе установки значения сигналов можно указать временную задержку, по

истечении которой следует присвоить сигналу результат вычисления соответствующего выражения, например:

```
SUM <= (A xor B) after 2 ns;
```

Кроме того, у сигнала есть еще одно отличие от переменной: он может представлять не одно значение, а несколько значений на своей временной диаграмме, используя семейство событий, например:

```
signal wavefrm : std_logic;
wavefrm <= '0', '1' after 5 ns, '0' after 10 ns, '1' after 20 ns;
```

Внимание! *Помните, что переменная изменяет свое значение сразу же после завершения выполнения оператора присвоения. Сигнал же изменяет свое значение после завершения выполнения оператора установки значения сигнала не сразу, а лишь по истечении заданной временной задержки. Если время задержки явно не задано, сигнал изменится после задержки* `delta`*, установленной по умолчанию. Последнее обстоятельство, т. е. возможность явно не указывать временную задержку в операторах установки значения сигналов, может привести к неверным результатам моделирования из-за неправильной трактовки разработчиком программного кода, в котором используются переменные и сигналы.*

Давайте рассмотрим два примера, в первом из которых используются переменные, а во втором — сигналы.

Пример 5.1. Процесс, использующий переменные

```
architecture VAR of EXAMPLE is
    signal TRIGGER, RESULT: integer := 0;
begin
  process
       variable variable1: integer :=1;
       variable variable2: integer :=2;
       variable variable3: integer :=3;
  begin
       wait on TRIGGER;
       variable1 := variable2;
       variable2 := variable1 + variable3;
       variable3 := variable2;
       RESULT <= variable1 + variable2 + variable3;
```

```
  end process;
end VAR
```

Пример 5.2. Процесс, использующий сигналы

```
architecture SIGN of EXAMPLE is
   signal TRIGGER, RESULT: integer := 0;
   signal signal1: integer :=1;
   signal signal2: integer :=2;
   signal signal3: integer :=3;
begin
    process
     begin
       wait on TRIGGER;
       signal1 <= signal2;
       signal2 <= signal1 + signal3;
       signal3 <= signal2;
       RESULT <= signal1 + signal2 + signal3;
  end process;
end SIGN;
```

В примере 5.1 операторы присвоения значений переменным `variable1`, `variable2` и `variable3` вычисляются последовательно, т. е. строка за строкой. При этом их значения модифицируются этими операторами мгновенно и становятся известными к моменту появления сигнала `TRIGGER`. Следовательно, сигнал `RESULT` вычисляется уже с новыми значениями переменных, а именно: `variable1 = 2`, `variable2 = 5 (=2+3)`, `variable3 = 5`. Так как сигнал `RESULT` будет вычислен во время активизации сигнала `TRIGGER`, то момент модификации его значения будет определяться установленной по умолчанию задержкой: `TRIGGER + delta`. Новое значение сигнала `RESULT` будет равно `RESULT=12`.

В примере 5.2 сигналы начнут вычисляться во время активизации сигнала `TRIGGER`. При вычислении новых значений сигналов в этот момент будут использоваться начальные значения сигналов. Все сигналы, участвующие в процессе вычислений, будут модифицированы с временной задержкой `delta` после того, как активизирован сигнал `TRIGGER`. Таким образом, сигналы будут иметь следующие значения: `signal1 = 2`, `signal2 = 4 (=1+3)`, `signal3 = 2` и `RESULT=7`.

ГЛАВА 6

ТИПЫ ДАННЫХ

Как уже отмечалось в предыдущих главах, в языке любой программный элемент данных, будь то константа, переменная или сигнал, относится к тому или иному конкретному типу. *Тип* (type) задает информационные характеристики программного элемента данных, определяет набор допустимых значений элемента, а также набор операций, которые применимы к этому элементу. Язык VHDL относится к разряду языков со строгим контролем типов, т. е. в VHDL не допускается использование нетипизированных данных, как, например, это имеет место в языке Basic. Более того, в VHDL не допускается также присвоение значения одного типа элементу данных другого типа, что являетя вполне легитимной операцией в таких языках программирования со строгим контролем типов, как C++ или Pascal. В C++, например, можно присвоить целочисленное значение переменной, имеющей действительный тип. В C++ при этом компилятор автоматически выполняет приведение типов, тогда как в VHDL такая операция присвоения является недопустимой: разработчик должен позаботиться о преобразовании типов до выполнения операции присвоения.

6.1. Встроенные типы данных пакета STANDARD

Спецификации языка VHDL предусматривают использование нескольких *встроенных типов* данных (**Табл. 6.1**), определенных в *пакете* `standard` библиотеки `std`. Для того чтобы использовать эти типы данных, в проект следует включить следующие операторы:

```
library std, work;
  use std.standard.all;
```

Таблица 6.1. **Типы данных, определенные в пакете std.standard**

Тип	Множество допустимых значений	Пример
bit	Принимает значения '0', '1'	`signal A : bit := '1';`
bit_vector	Массив элементов типа bit	`signal INBUS : bit_vector (7 downto 0);`
boolean	Принимает значения FALSE, TRUE	`variable TEST : Boolean := FALSE;`
character	Любой допустимый символ VHDL (печатные символы должны заключаться в одинарные кавычки, например '#')	`variable VAL : character := '$';`
file_open_kind	read_mode, write_mode, append_mode	
file_open_status	open_ok, status_error, name_error, mode_error	
integer	Диапазон допустимых значений зависит от реализации, но во всех реализациях обеспечивается диапазон не менее чем от $-(2^{31}-1)$ до $+(2^{31}-1)$, т. е. от –2147483648 до 2147483647	`constant CONST1: integer := 129;`
natural	Целочисленное значение от 0 и до максимального положительного значения целых чисел, обеспечиваемого конкретной реализацией	`variable VAR1 : natural := 2;`
positive	Целочисленное значение от 1 и до максимального положительного значения целых чисел, обеспечиваемого конкретной реализацией	`variable VAR2 : positive := 2;`
real	Число с плавающей точкой в диапазоне от -1.0×10^{38} до $+1.0 \times 10^{38}$ (диапазон зависит от реализации)	`variable VAR3 : real := +64.2E12;`
severity_level	Предупреждения, а также сообщения об ошибках и сбоях	
string	Массив, каждый элемент которого представляет собой единичный символ	`variable VAR4 : string (1 to 12) := "@$#ABC*()_%Z";`
time	Целочисленное значение, диапазон которого зависит от конкретной реализации; время может выражаться в секундах (sec), миллисекундах (ms), микросекундах (us), наносекундах (ns), пикосекундах (ps) и фемтосекундах (fs), а также в минутах (min) и часах (hr)	`variable DELAY : time := 5 ns;`

6.2. Пользовательские типы и подтипы

Разработчик может ввести в проект новые типы, используя оператор объявления типа, который начинается ключевым словом **type**, снабжается идентификатором и определяет диапазон значений для нового пользовательского типа. Синтаксис объявления типа имеет следующий вид:

```
type идентификатор_типа is range значение1 (to)/(downto)
значение2;
```

Кроме того, в VHDL допускается создание пользовательских *подтипов* (subtype). Оператор объявления подтипа начинается ключевым словом **subtype**, снабжается идентификатором и идентификатором родительского типа, а также диапазоном значений для нового пользовательского подтипа. Синтаксис объявления подтипа имеет следующий вид:

```
subtype идентификатор is идентификатор_типа  range значение1 to
(downto) значение2;
```

Подтип представляет собой подмножество значений, взятое из родительского типа. Подтипы удобно использовать в тех случаях, когда по тем или иным причинам в VHDL-проекте нужно использовать часть диапазона, определенного типом, но при этом создавать отдельный тип не обязательно.

Ниже приведены примеры операторов объявления пользовательских типов и подтипов:

```
type small_int is range 0 to 1024;
type my_word_length is range 31 downto 0;
subtype data_word is my_word_length range 7 downto 0;
subtype int_small is integer range -1024 to +1024;
type cmos_level is range 0.0 to 3.3;
type pmos_level is range -5.0 to 0.0;
type probability is range 0.0 to 1.0;
subtype cmos_low_V is cmos_level range 0.0  to +1.8;
```

Примеры операторов объявления переменных и сигналов с пользовательскими типами и подтипами таковы:

```
variable BUS_WIDTH : small_int := 24;
signal DATA_BUS : my_word_length;
variable VAR1: cmos_level := 1.48;
signal DATA_BUS12 : my_word_length;
```

6.2.1. Физические типы

Язык VHDL поддерживает работу с физическими величинами. Однако в пакете `std.standard` определены только единицы измерения времени. В терминах языка VHDL это означает, что в пакете `standard` объявлен только *физический тип* (physical type) **`time`**. Синтаксис определения этого типа имеет следующий вид:

```
type time is range -2147483647 to 2147483647
 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;
```

Для работы с другими физическими величинами разработчик должен сам определить для них единицы измерения, т. е. объявить пользовательский тип для представления в VHDL-проекте требуемой физической величины. Так, например, если пользователю требуется в проекте работать с единицами измерения сопротивления участка цепи, он должен определить физический тип, например с именем `conductance`. В этом случае оператор объявления типа `conductance` может иметь вид

```
type conductance is range 0 to 2E-9
   units
       mho;
       mmho = 1E-3 mho;
       umho = 1E-6 mho;
       nmho = 1E-9 mho;
```

```
        pmho = 1E-12 mho;
    end units conductance;
```

Ниже показан пример оператора объявления константы LINE_COND, имеющей пользовательский физический тип conductance:

```
constant LINE_COND: conductance:= 125 umho;
```

Внимание! *При сопровождении числового значения единицей измерения перед именем единицы измерения должен присутствовать пробел.*

В тех случаях, когда нужно многократно использовать однажды объявленный пользовательский тип, следует либо включить оператор объявления этого типа в раздел объявлений архитектуры проекта, либо поместить такой оператор в *пакет* (pachage). Последний подключается к проекту с помощью оператора **package**. Например, приведенный ниже пакет my_types можно подключить к проекту, воспользовавшись оператором **package** my_types;.

```
package my_types is
    type small_int is range 0 to 1024;
    type my_word_length is range 31 downto 0;
    subtype data_word is my_word_length is range 7 downto 0;
    type cmos_level is range 0.0 to 3.3;
    type conductance is range 0 to 2E-9
         units
              mho;
              mmho = 1E-3 mho;
              umho = 1E-6 mho;
              nmho = 1E-9 mho;
              pmho = 1E-12 mho;
         end units conductance;
end package my_types;
```

6.2.2. Перечислимые типы

Базовым компонентом *перечислимых типов* (enumerated type) является список литеральных символов и (или) идентификаторов. Язык VHDL поддерживает работу как со встроенным перечислимым типом, так и с пользовательскими перечислимыми типами. Последние очень

удобно использовать при формировании проектов на абстрактном уровне, т. е. проектов конечных автоматов. Синтаксис объявления перечислимого типа имеет следующий вид:

```
type имя_типа is (список_идентификаторов_или_символов);
```

Ниже приведено несколько примеров операторов объявлений перечислимых типов.

```
type my_3values is ('0', '1', 'Z');
type PC_OPER  is (load, store, add, sub, div, mult, shiftl,
shiftr);
type  hex_digit  is  ('0', '1', '2', '3', '4', '5', '6', '7', '8',
'9', 'A', 'B', 'C', 'D', 'E', 'F');
type state_type is (S0, S1, S2, S3);
```

Примеры операторов объявлений программных элементов данных с приведенными выше перечислимыми типами таковы:

```
signal  SIG1:  my_3values;
variable  ALU_OP :  pc_oper;
variable  first_digit : hex_digit := '0';
signal STATE : state_type := S2;
```

Примечание. *Если в операторе объявления сигнала с перечислимым типом явно не указывается начальное значение сигнала, по умолчанию используется первое (крайнее левое) значение из списка, указанного в объявлении соответствующего перечислимого типа.*

Примером встроенного перечислимого типа, объявленного в пакете `std_logic_1164`, является тип `std_ulogic`. Оператор объявления типа `std_ulogic` имеет вид

```
type STD_ULOGIC is (
   'U', -- uninitialized
   'X', -- forcing unknown
   '0', -- forcing 0
   '1', -- forcing 1
   'Z', -- high impedance
   'W', -- weak unknown
   'L', -- weak 0
```

```
'H', -- weak 1
'-'); -- don't care
```

Чтобы использовать этот тип в проекте, разработчик должен включить перед каждым объявлением интерфейса проекта следующую пару операторов:

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

Этот тип создан исходя из того, что в ЦУ в общем случае может быть несколько *формирователей сигнала*, называемых также *драйверами* (driver), между которыми может возникнуть конкуренция. В случае возникновения конфликтной ситуации между несколькими источниками выходной сигнал будет неопределенным. Простейший пример такой конфликтной ситуации — это соединение выходов логического элемента **and** и логического элемента NOT с подключением общего выходного сигнала к выводу OUT1. Для того чтобы разрешить конфликт между источниками сигнала и установить окончательное значение для сигнала на выводе OUT1, требуется вызвать так называемую *разрешающую функцию* (resolution function).

Так, если сигнал имеет тип std_ulogic и может формироваться несколькими драйверами, разработчик должен позаботиться о создании разрешающей функции. В пакете std_logic_1164 определена используемая по умолчанию разрешающая функция RESOLVED. Создав в VHDL-проекте пользовательскую функцию с таким именем, можно использовать следующий оператор объявления сигнала OUT1, формируемого двумя конфликтующими источниками:

```
signal OUT1: RESOLVED: std_ulogic;
```

Таким образом, если возникнет конкуренция нескольких формирователей сигнала, для разрешения конфликта будет вызвана функция RESOLVED, в которой разработчик должен разместить программный код, определяющий конечное значение сигнала OUT1. Можно также объявить сигнал OUT1 как сигнал типа std_logic, поскольку в операторе объявления подтипа std_logic, определенного в пакете std_logic_1164 на основе родительского типа std_ulogic, разрешающая функция RESOLVED используется по умолчанию. Это позво-

ляет разработчику в VHDL-проекте ограничиться объявлением сигнала следующего вида:

```
signal OUT1: std_logic;
```

Для разрешения конфликтов разработчику остается лишь создать соответствующий программный код функции RESOLVED.

6.2.3. Композитные пользовательские типы

Композитные пользовательские типы используются для объявления программных элементов данных (констант, переменных и сигналов), состоящих из нескольких простых типов. Полученные композитные программные элементы представляют собой либо *массив* (array), либо *запись* (record).

Массивы

Оператор объявления массива в языке VHDL выглядит следующим образом:

```
type имя_типа_массива is array (схема_индексации) of
тип_базового_элемента;
```

Ниже приведены примеры операторов объявления массивов.

```
type MY_WORD  is array  (15 downto 0)  of std_logic;
type YOUR_WORD  is array (0 to 15)  of std_logic;
type VAR  is array  (0 to 7)  of  integer;
type std_logic_1D is array (std_ulogic) of std_logic;
```

В первых двух примерах определены типы одномерных массивов MY_WORD и YOUR_WORD, каждый из которых состоит из 16 элементов типа std_logic. Индексация массива типа MY_WORD выполняется в нисходящем порядке (первый элемент массива — MY_WORD(15), последний — MY_WORD(0)), а массива типа YOUR_WORD — в восходящем (первый элемент массива — YOUR_WORD(0), последний — YOUR_WORD(15)).

В третьем примере определен тип одномерного массива VAR, содержащий 8 элементов типа integer.

В последнем примере определен тип одномерного массива std_logic_1D, содержащего элементы типа std_logic. Необычным в операторе объявления этого типа массива является то, что количество элементов такого массива определяется не числами, а типом std_ulogic. Поскольку тип std_ulogic — это перечислимый тип, состоящий из 9 элементов, массив типа std_logic_1D также имеет размер 9, а значением каждого элемента такого массива является символ, определяемый соответствующим элементом std_ulogic. Например, std_logic_1D('U') = 'U', std_logic_1D('X') = 'X', std_logic_1D('0') = '0' и т. д. до std_logic_1D('-') = '-'.

Объявив композитный тип массива, разработчик может использовать его в качестве пользовательского типа в операторе объявления программного элемента данных (константы, переменной или сигнала), например:

```
signal MEM_ADDR : MY_WORD;
signal DATA_WORD : YOUR_WORD :=  B"1101100101010110";
constant SETTING : VAR := (2, 4, 6, 8, 10, 12, 14, 16);
```

В первом операторе объявлен сигнал MEM_ADDR, представляющий собой 16-битный массив, инициализированный нулями. Сигнал DATA_WORD — это также 16-битный массив, но инициализированный битовой строкой. В программном коде VHDL-проекта к элементам массивов обращаются, используя идентификатор соответствующего программного элемента данных и индекс. Например, для обращения к первым элементам объявленных выше массивов следует использовать идентификаторы вида MEM_ADDR(15), DATA_WORD(0) и SETTING(0).

В VHDL можно также создать массив, представляющий собой фрагмент, или *сектор*, соответствующего пользовательского типа массива. Для этого необходимо в операторе объявления массива явно указать схему индексации, являющейся подмножеством схемы индексации базового типа массива: MEM_ADDR (15 **downto** 8), DATA_WORD (0 **to** 7) и т. п.

Кроме того, язык VHDL поддерживает работу не только с одномерными, но и с двухмерными массивами. Для объявления двухмерных типов массивов используются операторы **type** … **is array** … **of**, в схеме индексации которых указывается не один, а два диапазона индексов, например:

```
type MY_MATRIX3X2 is array (1 to 3, 1 to 2) of natural;
type YOUR_MATRIX4X2 is array (1 to 4, 1 to 2) of integer;
type std_logic_2D is array (std_ulogic, std_ulogic) of std_logic;
```

В последнем примере тип `std_logic_2D` представляет собой двухмерный массив размерностью 9×9 или таблицу, индексированную по столбцам и строкам элементами `std_ulogic`.

Для иллюстрации того, как в языке VHDL реализовано размещение элементов двухмерного массива в памяти, рассмотрим пример обьявления переменной-массива `DATA_ARR` типа `YOUR_MATRIX4X2`.

```
variable DATA_ARR : YOUR_MATRIX4X2 := ((0, 2), (1, 3), (4, 6), (5, 7));
```

Здесь переменная `DATA_ARR`, которая представляет собой двухмерный массив типа `YOUR_MATRIX4X2`, инициализирована парами значений. Первое значение каждой пары размещается в первом столбце соответствующей строки, а второе — во втором столбце этой же строки:

```
0 2
1 3
4 6
5 7
```

Для доступа к элементу массива требуется указать идентификатор переменной-массива с номерами строки и столбца, т. е. обращение к переменной вида `DATA_ARR(3,1)` вернет значение `4`.

Иногда при объявлении типа массива целесообразно не определять его размерность. В языке VHDL поддерживается такая возможность, что позволяет создавать массивы неограниченного размера, или *открытые массивы*. Синтаксис оператора объявления типа открытого массива имеет следующий вид:

```
type имя_типа_массива is array (тип range <>) of
тип_базового_элемента;
```

Ниже приведены примеры объявления типов открытых массивов.

```
type MATRIX is array (integer range <>)  of  integer;
type VECTOR_INT  is array (natural  range <>)  of  integer;
type VECTOR2 is array (natural range <>, natural range <>) of std_logic;
```

При объявлении программного элемента данных с типом открытого массива задается фактический размер массива, т. е. диапазоны изменения значений его индексов, например:

```
variable MATRIX8 : MATRIX (2 downto -8) := (3, 5, 1, 4, 7, 9, 12,
                                                14, 20, 18);
variable ARRAY3x2 : VECTOR2 (1 to 4, 1 to 3)) := (('1' ,'0'), ('0',
                                                  '-'), (1, 'Z'));
```

Записи

Как и массив, запись — это композитный пользовательский тип данных, содержащий несколько простых типов. Однако в отличие от массива, элементы записи в общем случае могут иметь разные типы. Синтаксис оператора объявления типа записи имеет следующий вид:

```
type имя_типа_записи is record
   идентификатор: тип;
   :
   идентификатор :тип;
end record;
```

На практике операторы объявления типа записи могут иметь различный вид, определяемый номенклатурой и типом входящих в запись базовых элементов, например:

```
type MY_MODULE is record
   RISE_TIME : time;
   FALL_TIME : time;
   SIZE      : integer range 0 to 200;
   DATA      : bit_vector (15 downto 0);
end record;
```

Объявив тип записи MY_MODULE, разработчик может создать программный элемент данных (константу, переменную или сигнал) с этим типом. Например, объявление сигналов типа MY_MODULE может иметь следующий вид:

```
signal A, B: MY_MODULE;
```

Для того чтобы в программном коде VHDL-проекта назначить значения нужному элементу записи, необходимо использовать одну из следующих форм обращений к таким элементам:

```
A.RISE_TIME <= 5 ns;     -- установить значение сигнала RISE_TIME
                         -- записи A типа MY_MODULE;
A.SIZE = 120;            -- присвоить значение элементу SIZE записи A
                         -- типа MY_MODULE;
B <= A;                  -- установить сигналу B типа MY_MODULE
                         -- значение сигнала A типа MY_MODULE.
```

Как видно из приведенных примеров, для обращения к элементу записи требуется указать общее имя записи и имя конкретного элемента записи, разделив эти имена точкой.

Записи очень полезны при создании проектов арифметико-логических устройств (АЛУ) персональных компьютеров (ПК) или других ЦУ, управляемых командами со сложной структурой, поскольку использование записей повышает наглядность и упрощает восприятие программного кода. Рассмотрим, например, следующий фрагмент подобного проекта:

```
type RegName is (AX, BX, CX, DX);  --  перечислимый тип
type Komand is record
Mnemonic : String (1 to 10);
Code_Oper : Bit_Vector(3 downto 0);
Operand1, Operand2, Adres : RegName;
end record;
```

Определенный в этом фрагменте тип записи `Komand` содержит информацию о списке команд процессора. В структуре объявленного типа `Komand` содержится 5 элементов, каждый из которых является фрагментом отдельной инструкции (например, инструкций `Instr1`, `Instr2`, `Instr3`, `Instr4`, ..., `InstrN`) программы, управляющей работой процессора ПК. Так, элемент `Mnemonic` типа `String` представляет собой мнемоническую часть кода инструкции. Элемент `Code_Oper` с типом `Bit_Vector` является кодом операции. Элементы `Operand1`, `Operand2`, `Adres` типа `RegName` являются соответственно операндами инструкции и адресом. Обратите внимание на то, что последние три элемента объявлены в одной строке, поскольку они имеют одинаковый тип.

Теперь, используя тип `Komand`, разработчик может в удобной для восприятия форме представить любую инструкцию, содержащуюся в программе, например:

```
variable Instr1, Instr2, Instr3: Komand;
. . .
Instr1:= ("ADD AX, BX", "0001", AX, BX, AX);
Instr2:= ("SUB AX, DX", "0010", AX, DX, BX);
Instr3:= ("MUL AX, DX", "0110", AX, DX, BX);
```

Доступ к элементам записей типа Komand и присвоение им значений можно организовать по такой схеме:

```
Instr3.Mnemonic := "MUL AX, DX";
Instr3.Code_Oper : = "0110";
Instr3.Operand2 : = DX;
```

Рассмотрим примеры использования записи.

Пример 6.1. Использование записи для хранения промежуточных результатов

```
library ieee;
use ieee.std_logic_1164.all;
entity Arith is
    port(A : in INTEGER range 0 to 256;
         B : in INTEGER range 0 to 256;
         Kod_Op : in std_logic_vector(1 downto 0);
         Result : out  INTEGER range 0 to 32563);
end Arith;
architecture Arith_ar of Arith is
   type prm_res is record
        Sum : INTEGER range 0 to 32563;
        Sub : INTEGER range 0 to 32563;
        Mul : INTEGER range 0 to 32563;
        Div : INTEGER range 0 to 32563;
   end record;


 signal buf : prm_res;


 signal Res_sum   : INTEGER range 0 to 32563;
   signal Res_sub : INTEGER range 0 to 32563;
   signal Res_Mul : INTEGER range 0 to 32563;
   signal Res_Div : INTEGER range 0 to 32563;
```

```
begin
   buf.Sum <= A + B;          Res_Sum <= buf.Sum;
   buf.Sub <= A – B;          Res_Sub <= buf.Sub;
   buf.Mul <= A * B;          Res_Mul <= buf.Mul;
   buf.Div <= A / B;          Res_Div <= buf.Div;


 Result <= Res_Sum when Kod_Op = "00" else
           Res_Sub when Kod_Op = "01" else
           Res_Mul when Kod_Op = "10" else
           Res_Div when Kod_Op = "11" ;
end Arith_ar;
```

Проект `Arith` (пример 6.1) моделирует работу универсального арифметического процессора, который параллельно выполняет набор арифметических операций над беззнаковыми целыми числами. Результаты своей работы, т. е. результаты параллельно выполненных операций, он сохраняет в элементах буфера `buf`, представляющего собой запись типа `prm_res`. В зависимости от значения кода операции `Kod_Op`, поступающего на входной порт, процессор пересылает на выходной порт `Result` только результаты одной операции.

Пример 6.2. Использование записи в качестве входных/выходных портов

```
library ieee;
use ieee.std_logic_1164.all;
library Work;
use Work.rec_package1.all;
entity rec is
   port(A: in my_rec;
        B: out  yo_rec);
end rec;
architecture rec_ar of rec is
begin
proc0 : process(A)
    begin
         if A.my_bit = '0' and A.my_int /= 2 then
            B.yo_bit <= '1';
            B.yo_char <= 'H';
            B.yo_int <= 5;
```

```
        elsif A.my_bit = '1' and A.my_int = 2 then
           B.yo_bit <= '0';
           B.yo_char <= 'S';
           B.yo_int <= 3;
        end if;
    end process;
end rec_ar;
```

Обратите внимание на то, что для корректной работы проекта `Rec` (пример 6.2) требуется определить пользовательский пакет с именем имя `rec_package1` и поместить его в библиотеку `Work`. В пакете должно храниться описание пользовательских типов `my_rec` и `yo_rec`, которые определяют типы входных/выходных портов.

Содержимое пакета может иметь следующий вид:

```
package rec_package1 is
   type my_rec is record
        my_bit : bit;
        my_char : character;
        my_int : integer range 0 to 15;
   end record;

   type yo_rec is record
        yo_bit : bit;
        yo_char : character;
        yo_int : integer range 0 to 15;
   end record;
end rec_package1;
```

6.3. Преобразование типов

Как уже отмечалось выше, язык VHDL относится к классу языков со строгим контролем типов. Причем, если в других языках этого класса допускается назначение программным элементам данных значений других типов, в VHDL такая операция является недопустимой. Данное ограничение распространяется как на сигналы, так и на константы и переменные. Более того, в VHDL запрещено также выполнять совместные операции над данными разных типов в одном выражении.

Однако на практике иногда необходимо использовать в одном выражении разнотипные программные элементы данных. Разрешить

проблему совместимости разных типов в одном выражении можно путем преобразования одного типа в другой (такую операцию называют также приведением типов). В других языках программирования приведение типов, как правило, выполняется автоматически компилятором. В VHDL разработчик должен позаботиться о соответствующих операциях самостоятельно, создавая нужный программный код по мере необходимости. В пакетах `std_logic_1164` и `std_logic_arith` библиотеки `ieee` имеется целый ряд функций преобразования типов, значительно упрощающих решение этой задачи. Функции пакета `std_logic_1164` приведены в **Табл. 6.2**.

Таблица 6.2. **Функции преобразования типов, содержащиеся в пакете std_logic_1164**

Функция	Тип аргумента	Тип результата
to_Bit	std_ulogic	bit
to_BitVector	std_logic_vector	bit_vector
to_BitVector	std_ulogic_vector	bit_vector
to_StdULogic	bit	std_ulogic
to_StdLogicVector	bit_vector	std_logic_vector
to_StdUlogicVector	bit_vector	std_ulogic_vector
to_StdLogicVector	std_ulogic	std_logic_vector
to_StdUlogicVector	std_logic	std_ulogic_vector

Кроме того, в пакетах `std_logic_unsigned` и `std_logic_vector` библиотеки `ieee` содержатся определения функций, позволяющих преобразовывать тип `integer` в `std_logic_vector` и наоборот.

Для достижения подключения функций преобразования типов в подраздел объявления библиотек и пакетов проекта нужно включить следующие операторы:

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

В примерах 6.3 и 6.4 проиллюстрированы методы организации приведения типов в VHDL, основанной на использовании функций преобразования типов программных элементов данных.

Пример 6.3. Преобразование типа bit_vector в тип std_logic_vector

```
library ieee;
use ieee.std_logic_1164.all;
entity QUAD_NAND2 is
   port (A, B: in bit_vector(3 downto 0);
         out4: out std_logic_vector (3 downto 0));
end QUAD_NAND2;
architecture behavioral_2 of QUAD_NAND2 is
begin
out4 <= to_StdLogicVector(A and B);
end behavioral_2;
```

В примере 6.3 проекта QUAD_NAND2 выражение A **and** B, которое имеет тип bit_vector, должно быть приведено к типу std_logic_vector, прежде чем может быть использовано для установки сигнала **out** 4, имеющего тип std_logic_vector.

Пример 6.4. Преобразование типа Unsigned в тип Integer

```
library ieee;
use ieee.std_logic_arith.all;
entity add_int is
port(A : in unsigned (7 downto 0) := b"11110001";
    B : in unsigned (7 downto 0) := b"10010001" ;
  res : out integer);
end add_int;
architecture conv of add_int is
begin
res <= conv_integer(A) + conv_integer(B);
end conv;
```

В примере 6.4 приведен проект сумматора add_int, входные данные которого A и B имеют тип unsigned, а результат res — тип integer. Несмотря на подобие типов integer и unsigned, прямое назначение результату суммы входных данных в VHDL является недопустимым. Поэтому с целью приведения типов входных данных к типу результата в проекте используется функция conv_integer, преобразующая тип unsigned в тип integer. Объявление этой функции хранится в пакете ieee.std_logic_arith.

6.4. Атрибуты

Атрибуты (attributes) — это специальный комплекс подпрограмм-функций, которые позволяют получить информацию о характеристиках программных объектов. В языке VHDL поддерживает несколько типов *встроенных* атрибутов, т. е. атрибутов, определенных в ядре языка. Некоторые из встроенных атрибутов для своей работы требуют наличия фактического параметра (аргумента). Предопределенные атрибуты всегда присоединяются в виде суффикса к идентификатору сигнала, переменной или типа, отделяясь от него символом апострофа (**'**), за которым следует идентификатор атрибута:

идентификатор_элемента_данных' идентификатор_атрибута

Такой синтаксис позволяет получать различную информацию о сигнале, переменной или типе. Кроме встроенных атрибутов, разработчик может определять пользовательские атрибуты, а затем назначать их именованным объектам, указывая при определении объекта возможные значения его атрибута.

6.4.1. Атрибуты сигналов

Атрибуты сигналов — это встроенный в ядро языка комплекс специальных подпрограмм, обеспечивающий возможность получать характеристики сигналов при моделировании работы проекта (**Табл. 6.3**).

Таблица 6.3. **Часто используемые атрибуты сигналов**

Атрибут	Назначение
`Event`	Возвращает значение `TRUE` типа `boolean`, если на сигнале произошло событие, и `FALSE` в противном случае
`Active`	Возвращает значение `TRUE` типа `boolean`, если произошла установка сигнала, и `FALSE` в противном случае
`Transaction`	Возвращает сигнал типа `bit`, значение которого переключается (от 0 на 1 или от 1 на 0) при каждой транзакции на сигнале, для которого считывается атрибут
`last_event`	Возвращает интервал времени, начиная с последнего события на сигнале
`last_active`	Возвращает интервал времени, начиная с последней транзакции на сигнале
`last_value`	Возвращает значение сигнала, которое он имел перед последним произошедшим на сигнале событием

(продолжение)

Атрибут	Назначение
delayed(T)	Возвращает сигнал, чье значение соответствует значению сигнала, для которого считывается атрибут, с задержкой T (T — необязательный параметр, заданное по умолчанию значение T=0)
stable(T)	Возвращает значение TRUE типа boolean, если в течение интервала T значение сигнала не менялось, и FALSE в противном случае (T — необязательный параметр, заданное по умолчанию значение T=0)
quiet(T)	Возвращает значение TRUE типа boolean, если в течение интервала T на сигнале не происходило никаких событий, и FALSE в противном случае (T — необязательный параметр, заданное по умолчанию значение T=0)

Пример применения атрибута event к сигналу CLOCK: **if** (CLOCK'event **and** CLOCK='1') then .... Это выражение проверяет факт появления переднего фронта на тактовом сигнале CLOCK. Чтобы узнать, сколько времени прошло с момента появления переднего фронта последнего импульса CLOCK, можно использовать атрибут last_event: CLOCK'last_event.

6.4.2. Скалярные атрибуты

Скалярные атрибуты (scalar attributes) — это встроенный в ядро языка комплекс специальных подпрограмм, обеспечивающий получение характеристик скалярных типов данных при моделировании работы проекта. Язык VHDL поддерживает несколько встроенных атрибутов скалярного типа, приведенных в **Табл. 6.4**.

Таблица 6.4. **Часто используемые скалярные атрибуты**

Атрибут	Назначение
left	Возвращает крайнее левое значение диапазона допустимых значений, определяемого скалярным типом
right	Возвращает крайнее правое значение диапазона допустимых значений, определяемого скалярным типом
low	Возвращает наименьшее значение из диапазона допустимых значений, определяемого скалярным типом
high	Возвращает наибольшее значение из диапазона допустимых значений, определяемого скалярным типом
ascending	Возвращает TRUE, если скалярный тип имеет возрастающий диапазон, и FALSE в противном случае
value(S)	Возвращает индекс элемента перечисления из его представления в форме символьной строки S
image(s)	Возвращает строку символов по индексу элемента S перечислимого типа

Далее приведены примеры использования скалярных атрибутов.

```
-- Объявление типов.
type conductance is range 1E-6 to 1E3
   units mho;
end units conductance;
type my_index is range 3 to 15;
type my_levels is (low, high, '5', dontcare, highZ);
-- Примеры использования атрибутов.
conductance'right           -- возвращает значение 1E3.
conductance'high            -- возвращает значение 1E3.
conductance'low             -- возвращает значение 1E-6.
my_index'left               -- возвращает значение 3.
my_index'value("5")         -- возвращает значение 5.
my_levels'left              -- возвращает значение low.
my_levels'low               -- возвращает значение low.
my_levels'high              -- возвращает значение highZ.
my_levels'image(dontcare)   -- возвращает значение "dontcare".
my_levels'value("dontcare") -- возвращает значение dontcare.
```

Технология использования скалярных атрибутов представлена в примере 6.5.

Пример 6.5. Технология использования скалярных атрибутов

```
library ieee;
use ieee.std_logic_1164.all;
entity skal_attr is
   port(out_Im  : out  string (1 to 4);
        out_Im5  : out  string(1 to 3));
end skal_attr;
architecture skal_attr_ar of skal_attr is
type mu_lev is (ttt, 'r', '5', dont, vira, maina, petr45);
type my_index is range 3 to 15456;
signal sig_val  : mu_lev := vira;
signal sig_lov  : mu_lev;
signal sig_high : mu_lev;
signal sig_index_Val : my_index;
signal sig_index_Val_Char : character;
signal sig_index_Value :my_index;
signal sig_value : mu_lev;
begin
```

```
   out_Im <= mu_lev'image(dont);
-- значение out _Im устанавливается равным строке "dont".
   out_Im5 <= mu_lev'image(5);
-- значение out _Im5 устанавливается равным '5'.
   sig_val <= mu_lev'low;
-- значение sig_val устанавливается равным vira.
   sig_lov <= mu_lev'left;
-- значение sig_lov устанавливается равным ttt.
   sig_high <= mu_lev'right;
-- значение sig_lov устанавливается равным petr45.
   sig_index_Val <= my_index'value(756);
-- значение sig_index_Val устанавливается равным 756.
   sig_index_Value <= my_index'value("14");
-- значение sig_index_Value устанавливается равным 14.
   sig_value <= mu_lev'value("petr45");
-- значение sig_value устанавливается равным petr45.
   sig_index_Val_Char <= character'value(65);
-- значение sig_index_Val устанавливается равным 65.
end skal_attr_ar;
```

Временная диаграмма работы проекта показана на **Рис. 6.1**.

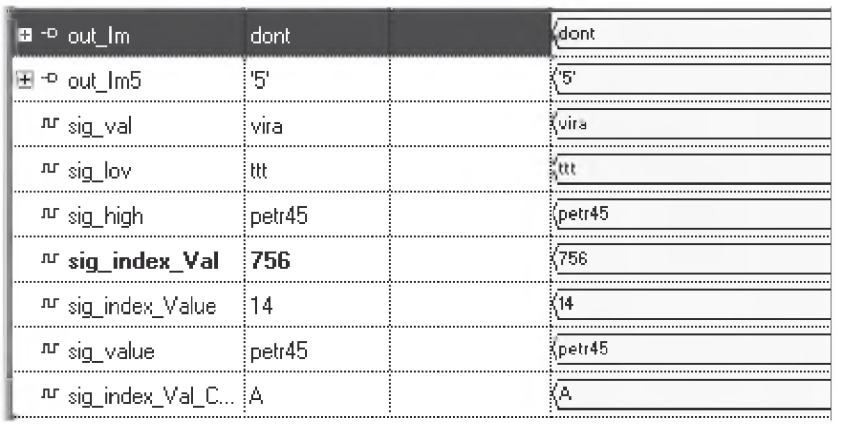

Рис. 6.1. Временная диаграмма работы проекта, приведенного в примере 6.5

6.4.3. Атрибуты массивов

Массивы в языке VHDL также могут иметь собственные атрибуты, часть которых представлена в **Табл. 6.5**. В частности, использование одного из видов *атрибутов массивов* (array attributes) позволяет получить параметры соответствующего диапазона, определяемого индексами соответствующего типа массива.

Таблица 6.5. **Часто используемые атрибуты массивов**

Атрибут	Назначение
`left(N)`	Возвращает значение левого индекса
`right(N)`	Возвращает значение правого индекса
`high(N)`	Возвращает правую верхнюю границу индекса
`low(N)`	Возвращает левую нижнюю границу индекса
`length(N)`	Возвращает количество элементов в индексе
`range(N)`	Возвращает диапазон массива
`reverse_range(N)`	Возвращает обращенный диапазон массива
`ascending(N)`	Возвращает значение `TRUE` типа `boolean`, если индекс имеет возрастающий диапазон, и `FALSE` в противном случае

Число `N`, стоящее в круглых скобках после идентификатора атрибута, обозначает размерность массива, по которой опрашивается атрибут. Для одномерного массива, число `N` всегда равно `1`, поэтому его можно опустить, как ниже показано в примерах. Атрибуты, приведенные в **Табл. 6.5**, можно применять и к двухмерным массивам:

```
type MYARR8x4 is array (8 downto 1, 0 to 3) of boolean;
type MYARR1 is array (-2 to 4) of integer;
MYARR1'left           -- возвращает значение -2.
MYARR1'right          -- возвращает значение  4.
MYARR1'high           -- возвращает значение  4.
MYARR1'reverse_range  -- возвращает диапазон  4 downto -2.
MYARR8x4'left(1)      -- возвращает значение  8.
MYARR8x4'left(2)      -- возвращает значение  0.
MYARR8x4'right(2)     -- возвращает значение  3.
MYARR8x4'high(1)      -- возвращает значение  8.
MYARR8x4'low(1)       -- возвращает значение  1.
MYARR8x4'ascending(1) -- возвращает значение False.
```

ГЛАВА 7

ОПЕРАЦИИ И СИМВОЛЫ ОПЕРАЦИЙ

В языке VHDL, как и в других языках программирования, поддерживаются различные *операции*, подразделяемые на семь основных классов (**Табл. 7.1**). Операции записываются с помощью *символов операций* (operator), которые применяются к двум (бинарные операции) или к одному (унарные операции) *операнду* (operand), в роли которых могут использоваться сигналы, переменные и константы. Как уже отмечалось выше, в VHDL в отличие от других языков программирования бинарные операнды обязательно должны иметь одинаковый тип.

Таблица 7.1. **Классы операций языка VHDL и символы операций**

Класс	Символы операций					
1. Логические операции	`and`	`or`	`nand`	`nor`	`xor`	`xnor`
2. Операции отношений	`=`	`/=`	`<`	`<=`	`>`	`>=`
3. Операции сдвига	`sll`	`srl`	`sla`	`sra`	`rol`	`ror`
4. Операции сложения	`+`	`=`	`&`			
5. Унарные операции	`+`	`-`				
6. Операции умножения	`*`	`/`	`mod`	`rem`		
7. Вспомогательные операции	`**`	`abs`	`not`			

Наиболее высоким приоритетом обладают операции 7-го класса, за ними идут операции 6-го класса и т. д., вплоть до операций 1-го класса, приоритет которых самый низкий. Если круглые скобки не используются, то сначала выполняются операции с более высоким приоритетом. Операции одного класса имеют одинаковый приоритет и выполняются слева направо по ходу записи выражения. Для примера рас-

смотрим порядок выполнения операций над векторами типа `std_ulogic_vectors` (`X='010'`, `Y='10'` и `Z='10101'`), связанных выражением **`not`** `X` **`&`** `Y` **`xor`** `Z` **`rol`** `1`. Это выражение эквивалентно выражению `((`**`not`** `X) &` `Y)` **`xor`** `(Z` **`rol`** `1) = ((101) &` `10)` **`xor`** `(01011) =(10110)` **`xor`** `(01011) = 11101`. Операция **`xor`** выполняется побитно.

7.1. Логические операции

Логические операции осуществляются с операндами типов `bit`, `boolean`, `std_logic` и `std_ulogic`, а также соответствующих векторов. Они используются, чтобы определять логические выражения (выражения булевой алгебры) и выполнять побитные операции на массивах битов. Результат имеет тот же тип, что и операнды. Эти операции могут применяться к сигналам, переменным и константам.

Обратите внимание на то, что операции **`nand`** и **`nor`** не ассоциативны, в связи с чем в последовательностях нескольких операций **`nand`** или **`nor`** нужно использовать круглые скобки, чтобы предотвратить синтаксическую ошибку. Например, попытка компиляции выражения

```
X nand Y nand Z
```

приведет к возникновению синтаксической ошибки. Для того чтобы исправить ее, следует переписать выражение в виде

```
(X nand Y) nand Z
```

7.2. Операции отношений

Операции отношений проверяют отношения значений двух операндов одинакового типа и возвращают результат `TRUE` или `FALSE`. Операции проверки равенства или неравенства можно выполнять над двумя операндами любых типов, лишь бы этот тип был одинаков у обоих операндов. Остальные операции можно выполнять только с операндами скалярного или дискретного типа.

Внимание! *Обратите внимание на то, что символ операции меньше или равно (<=) выглядит так же, как символ операции установки значения сигнала.*

В приведенных ниже примерах (строки 5, 6, и 7) первый левый символ <= обозначает операцию установки значения сигнала. В выражениях примеров (строки 5, 6, и 7) иллюстрируются символы операций отношений.

```
variable STS         : Boolean;
constant A           : integer :=24;
constant B_COUNT     : integer :=32;
constant C           : integer :=14;
STS <= (A < B_COUNT) ;                -- STS устанавливается
                                         значение "TRUE".
STS <= ((A >= B_COUNT) or (A > C));   -- Значение STS - "TRUE".
STS <= (std_logic ('1', '0', '1') < std_logic('0', '1', '1'));
                                      --  значение STS - "FALSE".
type new_std_logic is ('0', '1', 'Z', '-');
variable A1: new_std_logic := '1';
variable A2: new_std_logic := 'Z';
STS <= (A1 < A2);    -- Значение STS - "TRUE", поскольку символ '1'
                     -- меньше, чем символ 'Z'.
```

Обратите внимание на то, что операнды дискретного типа (например, перечислимого типа) сравниваются поэлементно, начиная слева направо.

7.3. Операции сдвига

Операции сдвига выполняют побитный или циклический сдвиг на одномерных массивах элементов типа `bit`, `std_logic` или `Boolean`. Эти операции являются бинарными. Левым операндом является одномерный битовый массив, массив с элементами типа `std_logic` или массив, содержащий элементы типа `Boolean`. Правый операнд должен быть целым числом (числом типа `integer`). Значение правого операнда определяет количество сдвиговых операций (**Табл. 7.2**). Тип результата определяется типом левого операнда.

Таблица 7.2. **Описание операций сдвига**

Операция	Описание
`sll`	Левый логический сдвиг. Правый бит заполняется нулями
`srl`	Правый логический сдвиг. Левый бит заполняется нулями
`sla`	Левый арифметический сдвиг. Правый бит сохраняет предыдущее значение
`sra`	Правый арифметический сдвиг. Левый бит сохраняет предыдущее значение
`rol`	Левый циклический сдвиг
`ror`	Правый циклический сдвиг

В приведенном ниже примере в результате выполнения операции **`srl`** над переменной `NUM1` последней будет присвоено значение: `"00100101"`.

```
variable NUM1 : bit_vector (7 downto 0) := "10010110";
variable NUM2 : bit_vector (7 downto 0);
NUM2 := NUM1 srl 2;
```

Сдвиговые операции обратимы. Это означает, что, когда правый операнд при выполнении сдвига влево является отрицательным числом, результат операции эквивалентен сдвигу вправо. Например, вместо операции `NUM2 := NUM1 srl 2;` в приведенном выше фрагменте можно было бы записать `NUM2 := NUM1 sll -2;` и получить такой же результат.

Ниже приведены другие примеры сдвиговых операций для переменной `A` типа `bit_vector` с начальным значением `A = "101001"`.

```
variable A: bit_vector := "101001";
A = A sll 2; -- A = "100100";
A = A srl 2; -- A = "001010";
A = A sla 2; -- A = "100111";
A = A sra 2; -- A = "111010";
A = A rol 2; -- A = "100110";
A = A ror 2; -- A = "011010";
```

7.4. Операции сложения

Операции сложения используются для выполнения арифметических действий (сложение и вычитание) над операндами любого числового

типа. Операция конкатенации (&) используется для объединения двух векторов, в результате чего получается более длинный вектор. Чтобы использовать эти операции, необходимо в дополнение к пакету ieee.std_logic_1164 подключить к проекту пакеты ieee.std_logic_unsigned.all или std_logic_arith.

Ниже приведен пример конкатенации группы объявленных сигналов в одну шину, роль которой на программном уровне играет сигнал MYBUS.

```
signal MYBUS        : std_logic_vector (15 downto 0);
signal STATUS       : std_logic_vector (2 downto 0);
signal RW, CS1, CS2 :std_logic;
signal MDATA        : std_logic_vector (0 to 9);
MYBUS <= STATUS & RW & CS1 & SC2 & MDATA;
```

В первой строке следующего примера выполняется конкатенация первых 8 единичных битов с 8 младшими битами массива MDATA с присвоением результата конкатенации массиву MYARRAY. Во второй строке выполняется конкатенация строк "VHDL" и "93" с присвоением результата массиву NEWWORD.

```
MYARRAY (15 downto 0) <= "1111_1111" & MDATA (2 to 9);
NEWWORD <= "VHDL" & "93";
```

7.5. Унарные операции

Унарными операциями называются операции, выполняемые над одним операндом. К таким операциям относятся операции изменения знака («+» или «–») операнда любых числовых типов.

7.6. Операции умножения

Операции умножения используются для создания математических выражений, содержащих операнды числовых типов (как целых, так и с плавающей точкой). Особенности использования операций умножения сводятся к учету типа правого операнда (**Табл. 7.3**).

Таблица 7.3. **Описание операций умножения**

Операция	Описание	Тип левого операнда	Тип правого операнда	Тип результата
*	Умножение	Любой целый тип или тип с плавающей точкой	Тот же тип, что и у левого операнда	Тот же тип, что и у левого операнда
		Любой физический тип	Тип **integer** или **real**	Тот же тип, что и у левого операнда
		Тип **integer** или **real**	Любой физический тип	Тот же тип, что и у правого операнда
/	Деление	Любой целый тип или тип с плавающей точкой	Любой целый тип или тип с плавающей точкой	Тот же тип, что и у левого операнда
		Любой физический тип	Тип **integer** или **real**	Тот же тип, что и у левого операнда
		Любой физический тип	Тот же тип, что и у левого операнда	**integer**
mod	Модуль (деление нацело)	Любой целый тип		Тот же тип, что и у левого операнда
rem	Остаток от деления	Любой целый тип		Тот же тип, что и у левого операнда

Операции нахождения остатка от деления (**rem**) и нахождения модуля (**mod**) определяются следующим образом:

```
A rem B = A -(A/B)*B    -- A/B - целое число.
A mod B = A - B * N     -- N - целое число.
```

Результат операции **rem** имеет знак его первого операнда, в то время как результат операции **mod** имеет знак второго операнда.

Примеры результатов этих операций показаны ниже, это:

```
A = 11 rem 4      -- A = 3
A = (-11) rem 4   -- A = -3
A = 9 mod 4       -- A = 1
A = 7 mod (-4)    -- A = -1, поскольку 7 - 4*2 = -1.
```

7.7. Вспомогательные операции

Операции нахождения абсолютного значения (**abs**) и возведения в степень (******) могут применяться к числовым типам. Логическое отри-

цание (`not`) создает инверсию от входной величины битового типа. Результат операции имеет тот же тип, что и тип левого операнда.

Следует заметить, что не все САПР поддерживают операцию возведения в степень в полном объеме. Как правило, в САПР реализуется лишь возможность возводить левый операнд во вторую степень.

7.8. Символы комментария

В языке VHDL *символы комментария* (comment operators) играют важную роль, поскольку с их помощью поддерживается система документирования исходного кода проектов пояснениями и примечаниями разработчика, называемыми комментариями. Наличие комментариев к проектам и субпроектам значительно облегчает процедуру создания комплексных проектов, так как любой участник рабочей группы проекта может из комментария выяснить характеристики проекта, интерфейсы его компонентов, требования к энергопотреблению и т. д. Наличие комментариев позволяет также распределить работу над проектом в рабочей группе, что обычно значительно сокращает сроки создания окончательного варианта проекта.

Как вы уже знаете, в VHDL символы комментария представляются двойным дефисом (--). Это означает, что комментарии в VHDL однострочные, т. е. область действия символов комментария в VHDL-проекте распространяется только на одну строку правее символа комментария. Следовательно, если разработчику необходимо создать многострочный комментарий, ему придется разместить символы комментария в начале каждой пояснительной записки.

С другой стороны, свойство распространения области действия символов комментария от той позиции, где они находятся, и до конца строки позволяет разработчику сопровождать комментариями операторов, располагая комментарии правее после операторов, как было неоднократно показано в приведенных выше примерах.

Все, что находится в строке правее символа комментария, игнорируется компилятором и не принимает никакого участия в процессах вычислений и моделирования работы проекта.

Следует отметить, что комментарии играют также важную роль в процессах отладки проекта. Известно, что проект с заданной функциональностью можно сформировать с помощью нескольких различных

наборов операторов. При этом каждый набор влияет на те или иные характеристики проекта, что требует от разработчика выбора оптимального набора путем тестирования имеющихся вариантов.

Сформировав несколько таких вариантов проекта, разработчик может поочередно протестировать каждый из них, комментируя (т. е. размещая символы комментария перед операторами) одни варианты и убрав эти символы перед операторами тестируемого варианта.

Ниже приведен пример VHDL-проекта, содержащего комментарии.

```
-- Проект моделирует работу криптогафического процессора
-- построенного на базе операций логического сдвига влево
-- и логического сдвига вправо.
-- Проект разработан 10.10.1992 года.
-- Автор проекта: Ковтанюк Юрий Славович
-- Copyright © by Yuriy Kovtanyuk. All rights reserved.
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee; -- Подключение к проекту библиотек и пакетов.
use ieee.std_logic_1164.all;
entity bit_shift is -- Объявление интерфейса проекта.
    port(
         clk : in std_logic;
         data_bit_in : in BIT_Vector (7 downto 0);
         kluch : in INTEGER range 0 to 7;
         shifr_data_bit : out  BIT_Vector(7 downto 0);
         deshifr_data_bit : out  BIT_Vector(7 downto 0));
end bit_shift;
--  Объявление архитектуры проекта.
architecture bit_shifta of bit_shift is
--  signal da_shi : BIT_Vector(7 downto 0); -- закомментированный
                                            -- оператор.
begin
   process (clk)
     -- Объявления оператора process.
     variable  data_in_prom, data_prom_Sdv_righ : BIT_Vector(7
                                                        downto 0);
   begin
     -- Выполняемые операторы оператора process.
        data_in_prom :=  data_bit_in;
     -- Шифрование данных.
```

```
        data_in_prom := data_in_prom sll kluch;
        shifr_data_bit <= data_in_prom ;
     -- Пересылка зашифрованных данных на выходной порт.
     -- Дешифрование.
        data_prom_Sdv_righ := data_in_prom srl kluch;
        deshifr_data_bit <= data_prom_Sdv_righ;
    end process;
end bit_shifta;
```

7.9. Математические выражения

Математические *выражения* (expression) — это важнейшие конструктивные элементы языка VHDL, которые дают в форме, наиболее приближенной к обычной математической записи, выполнять вычисления значений программных элементов данных проекта. Иными словами, выражение — это запись, которая определяет способ, с помощью которого вычисляются значения, состав операндов, состав операций и порядок их расположения в записи. Порядок, в котором выполняются операции над операндами, зависит от приоритета операций. Если операции, содержащиеся в выражении, имеют одинаковый приоритет, то по умолчанию вычисления производятся слева направо. Если требуется изменить порядок вычислений, следует применить круглые скобки.

Математические выражения входят в состав операторов присваивания, инициализации значений и установки значений сигналам. Они также являются конструктивными элементами других операторов языка VHDL (операторов **if**, **for** и т. д.).

Математические выражения в программировании принято делить на такие классы:

- арифметические;
- логические;
- логико-алгебраические.

Арифметическим принято называть выражение, содержащее такие операнды, как идентификаторы констант, переменных и сигналов, а также имена математических функций и функций преобразования типов, которые объединены знаками арифметических операций (**+**, **-**, *****, **/**, ******, **mod**, **rem**, операторы сдвига). Тип вычисленного значения выражения зависит от основного операнда выражения (как правило, левый операнд).

Логическим называют выражение, содержащее такие операнды, как идентификаторы констант, переменных и сигналов, имеющих тип `bit`, `bit_vector`, `std_logic`, `std_logic_vector`, `Boolean`, которые объединяются символами логических операций с использованием символа вспомогательной операции **`not`**. Такие выражения часто называют булевыми выражениями (Boolean expression). Тип вычисленного значения выражения зависит от типа основного операнда выражения.

Логико-алгебраические выражения содержат некоторый набор арифметических выражений, объединенных символами операций отношений, а также символами арифметических и (или) логических операций. Тип вычисленного значения логико-алгебраических выражений всегда является типом `boolean` (значения `TRUE` или `FALSE`).

Примеры выражений языка VHDL:

- `a + b - 2 * c + conv_integer (alpha)` — арифметическое выражение, в котором переменные `a`, `b`, `c` имеют тип `integer`, а переменная `alpha` — тип `signed`.
- `a - 35 >= (2 * c + 47) / 23**2` — логико-алгебраическое выражение.
- `m` **`and`** `s` — логическое выражение; `m` и `s` — переменные типа `bit`, `bit_vector std_logic` или `std_logic_vector`.

ГЛАВА 8

ПОВЕДЕНЧЕСКАЯ ФОРМА ПРОЕКТА: ЯВНО ЗАДАННЫЙ ОПЕРАТОР PROCESS

Как говорилось в гл. 2, язык VHDL поддерживает две формы представления ЦУ: поведенческую и структурную. Далее мы рассмотрим конструкции последовательных операторов языка VHDL, называемых также операторами регистровой логики, на использовании которых основывается поведенческая форма представления проектов ЦУ.

Последовательные операторы подчинены оператору `process`. При этом сам `process` имеет двоякую природу. С одной стороны, он относится к классу операторов параллельной обработки, что дает возможность использовать его для проектирования ЦУ комбинаторного типа. С другой стороны, он является родительским оператором для всех других выполняемых операторов, которые разрешено располагать в его теле. Часто говорят, что все выполняемые операторы, располагаемые в теле оператора `process`, выполняются по последовательному принципу. Эта особенность дает возможность применять это свойство оператора для проектирования ЦУ регистрового типа.

Оператор `process` может использоваться как в явном виде, так и в неявном. Об особенностях реализации VHDL-проектов в поведенческой форме на основе последнего мы поговорим в следующей главе, а в этой главе разберем особенности использования оператора `process`, задаваемого в явном виде.

8.1. Явно заданный оператор PROCESS

Явно заданный оператор `process` — это основная конструкция для поведенческой формы описания проектов, которая позволяет использовать в его теле последовательные операторы языка для описания

поведения проектируемого ЦУ регистрового типа на некотором временном интервале.

В языке VHDL различают два подкласса явно заданного оператора **process**:

- явно заданный оператор **process()** со списком чувствительности;
- явно заданный оператор **process** без списка чувствительности.

Синтаксис явно заданного оператора **process** имеет следующий вид:

```
[метка_процесса:] process [ (список_чувствительности) ] [is]
          [ операторы_объявлений_процесса ] -- Раздел объявлений.
begin -- Раздел выполняемых операторов.
   -- В этом разделе размещаются операторы следующих типов:
   -- установки значений сигналов;
   -- присвоения значения переменным;
   -- вызовы процедур;
   -- операторы case;
   -- операторы exit;
   -- операторы if;
   -- операторы loop;
   -- операторы next;
   -- операторы null;
   -- операторы wait;
end process [метка_процесса];
```

Примечание. *В конструкции приведенного выше синтаксиса оператора* ***process*** *в квадратных скобках указаны факультативные конструктивные элементы.*

Давайте рассмотрим пример использования оператора **process** в проекте D-триггера с асинхронной очисткой, переключающегося по переднему фронту синхроимпульса CLK:

```
library ieee;
use ieee.std_logic_1164.all;
entity DFF_CLEAR is
   port (CLK, CLEAR, D : in std_logic;
         Q : out std_logic);
end DFF_CLEAR;
architecture BEHAV_DFF of DFF_CLEAR is
```

```
begin
DFF_process: (CLK, CLEAR)
   begin
     if (CLEAR = '1') then
         Q <= '0';
     elsif (CLK'event and CLK = '1') then
            Q <= D;
     end if;
   end process;
end BEHAV_DFF;
```

Сам явно заданный оператор **process** располагается в разделе выполняемых операторов параллельной обработки, поскольку относится, как было сказано выше, к операторам соответствующего класса. Однако операторы, расположенные внутри оператора **process**, выполняются последовательно. Подобно другим операторам параллельной обработки, оператор **process** может считывать значения из входных портов и устанавливать значения сигналов на выходных портах интерфейса, а также взаимодействовать с остальными объектами архитектуры проекта. Иными словами, внутри оператора **process** можно устанавливать сигналы, определенные как внешние по отношению к этому оператору (в частности, на портах интерфейса, каким является, например, выход Q триггера рассматриваемого примера). Выражение `CLK'event and CLK = '1'` проверяет наличие переднего фронта сигнала CLK.

Список чувствительности (sensitivity list), который может использоваться в объявлении явного оператора **process**, — это набор сигналов, изменение значения которых вызывает немедленный запуск на выполнение оператора **process**. Если список чувствительности в операторе **process** не определен, внутри такого оператора **process** должен содержаться оператор **wait**, чтобы предоставить разработчику возможность управлять запуском и остановкой соответствующего оператора **process**. Оператор **process** без списка чувствительности всегда запускается на выполнение автоматически (без анализа активности каких-либо сигналов) в момент начала процесса моделирования работы проекта.

Внимание! *Одновременное использование в конструкции оператора* ***process*** *и списка чувствительности, и оператора* **wait** *недопустимо.*

Переменные и константы, которые используются в теле оператора **process**, должны быть определены в разделе объявлений оператора, т. е. перед ключевым словом **begin**. Ключевое слово **begin** сообщает о начале выполняемой части оператора **process**. Операторы, которые могут располагаться в этом разделе, выполняются последовательно, в соответствии с алгоритмом проекта.

Необходимо отметить, что присвоение значения переменной в теле оператора **process** выполняется немедленно и обозначено символами «:=». Операция присвоения значения переменной отличается от операции установки значения сигнала, обозначенной «<=», поскольку значение сигнала изменяется после заданного времени задержки. В отличие от сигналов измененное значение переменной доступно немедленно всем последовательным операторам, расположенным в пределах того же самого оператора **process** (*см.* примеры 5.1 и 5.2).

Приведенный выше пример проекта D-триггера поясняет, как создать последовательную форму VHDL-проекта на основе оператора **process**. Тем не менее, хотя оператор **process** главным образом используется для описания последовательных схем, с его помощью можно также описывать комбинационные схемы ЦУ. В частности, в приведенном ниже примере проекта полного однобитного сумматора, состоящего из двух полусумматоров, показано, как один оператор **process** может генерировать сигналы, указанные в списках чувствительности других операторов **process** и тем самым запускающие эти операторы на выполнение. Булевы функции, описывающие работу полусумматора и полного однобитного сумматора, имеют следующий вид.

- Описание полусумматора:

$$S_ha = (A \oplus B);\ C_ha = A \textbf{ and } B.$$

- Описание полного однобитного сумматора:

$$Sum = (A \oplus B) \oplus Cin = S_ha \oplus Cin;$$

$$Cout = (A \oplus B)\ Cin + AB = S_ha.Cin + C_ha,$$

где $\oplus$ — операция исключающего ИЛИ, представляемая в VHDL-проектах ключевым словом **xor**.

На **Рис. 8.1** показана структурная схема полного однобитного сумматора, состоящая из двух полусумматоров.

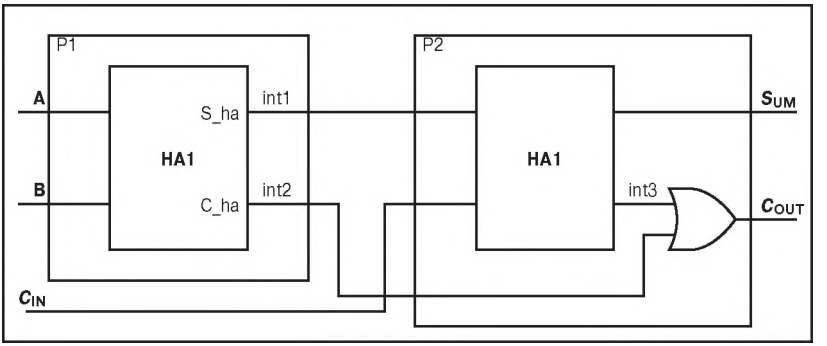

Рис. 8.1. Схема полного однобитного сумматора

Проект полного однобитного сумматора

```
library ieee;
use ieee.std_logic_1164.all;
entity FULL_ADDER is
   port (A, B, Cin : in std_logic;
         Sum, Cout : out std_logic);
end FULL_ADDER;
architecture BEHAV_FA of FULL_ADDER is
signal int1, int2, int3: std_logic;
begin
-- Процесс P1, определяющий первый полусумматор.
P1: process (A, B)
   begin
     int1   <= A xor B;
     int2   <= A and B;
   end process;
-- Процесс P2, определяющий второй полусумматор и вентиль OR.
P2: process (int1, int2, Cin)
   begin
     Sum    <= int1 xor Cin;
     int3   <= int1 and Cin;
     Cout   <= int2 or int3;
   end process;
end BEHAV_FA;
```

Примечание. *Строго говоря, в приведенном выше примере программы можно обойтись и одним оператором* **`process`**.

8.2. Оператор условной передачи управления If

Оператор **if** относится к семейству последовательных операторов (операторов регистровой логики), располагаемых в разделе выполняемых операторов явно заданного оператора **process()**.

Оператор **if** ответствен за выполнение того или иного блока последовательных операторов, располагаемых в его теле. Выбор для выполнения конкретного блока зависит от истинности одного или нескольких *условий* (condition). Синтаксис оператора **if** имеет следующий вид:

```
if условие1 then
   блок_операторов1;
[elsif условие2 then
   блок_операторов2; ]
. . .
[else
   блок_операторов; ]
end if;
```

Как видно из синтаксиса, оператор **if** является составным оператором. Он относится к подклассу операторов, допускающих вложенность, т. е. он может содержать новые операторы **if** или несколько дочерних конструкций **elsif**. При этом дочерняя конструкция **else** в структуре оператора **if** может быть только одна. Тело оператора «закрывается» словами **end if**. Когда в структуре оператора **if** используется много конструкций **elsif**, то тело каждой такой конструкции «закрывается» (заканчивается) соответствующими конструкциями **end if**. Количество **end if** должно соответствовать количеству вложенных **elsif** плюс 1.

Каждое условие является булевым выражением, т. е. выражением, значение которого имеет тип `boolean`. При выполнении оператор **if** проверяет каждое вложенное условие (вложенные операторы **if**, **elsif**) в том порядке, как они представлены в его теле, пока не будет найдено первое условие со значением `TRUE`. Как только такое условие найдено, выполнение передается тому блоку последовательных операторов, который соответствует данному условию.

Оператор **if** должен располагаться в подразделе выполняемых операторов оператора **process**.

Пример конструкции оператора **if** приводился выше в проекте D-триггера с асинхронной установкой. Оператор **if** может использоваться также для описания комбинационных устройств. Поясним это на следующем примере проекта однобитного четырехканального мультиплексора 4×1 со входами данных A, B, C и D, входами выбора канала S0 и S1 и выходом Z.

Примечание. *Как будет показано далее, для проектирования подобных комбинационных устройств удобнее применять не оператор* ***if****, а другие конструкции, такие как условная установка значения сигнала, операторы* ***when*** *...* ***else*** *и* ***when*** *...* ***select****.*

```
entity MUX_4_1a is
port (S1, S0, A, B, C, D: in std_logic;
    Z: out std_logic);
end MUX_4_1a;
architecture behav_MUX41a of MUX_4_1a is
begin
P1: process (S1, S0, A, B, C, D)
begin
  if ((not S1 and not S0)= '1') then
      Z <= A;
  elsif ((not S1 and S0) = '1') then
        Z<=B;
  elsif ((S1 and not S0) = '1') then
        Z <=C;
  else
        Z<=D;
  end if;
end process P1;
end behav_MUX41a;
```

Этот же мультиплексор можно также реализовать с помощью оператора **if** в следующем виде:

```
if S1='0' and S0='0' then
  Z <= A;
elsif S1='0' and S0='1' then
  Z <= B;
elsif S1='1' and S0='0' then
  Z <= C;
```

```
elsif S1='1' and S0='1' then
  Z <= D;
end if;
```

Примечание. *Оператор* **if** *часто используется при разработке VHDL-проектов конечных автоматов (КА).*

8.3. Оператор выбора CASE

Оператор выбора **case** *всегда* выполняет только один блок последовательных операторов из нескольких имеющихся в теле оператора блоков на основе значения задающего выражения. Выбирается тот блок, у которого сопровождающее значение тождественно равно текущему значению задающего выражения. Оператор выбора **case** имеет следующий синтаксис:

```
case задающее_выражение is
   when значение1 =>
        блок_операторов1;
   when значение2 =>
        блок_операторов2;
    -- в любом блоке операторов допускается
    -- использование вложенных операторов case.
   [ when others => блок_операторов ]
end case;
```

Значение задающего выражения должно иметь целый или перечислимый тип либо представлять собой одномерный массив битов. При выполнении оператор **case** сравнивает текущее вычисленное значение выражения с каждым указанным в теле оператора значением. Когда сравниваемые значения тождественно равны, происходит выбор соответствующего блока последовательных операторов. При написании операторов **case** должны соблюдаться следующие правила:

- в одном операторе нельзя указывать два блока с одинаковыми значениями выбора;
- если в операторе **case** отсутствует конструкция **when others**, *все* возможные значения задающего выражения должны быть представлены соответствующими значениями выбора.

Ниже приведен пример проекта, основанного на использовании оператора **case**, в котором в качестве задающего выражения используется целочисленный сигнал VALUE. В этом проекте реализуется следующая логика:

- выходной порт D = '1', когда сигнал VALUE имеет значение, лежащее между 51 и 60;
- выходной порт C = '1', когда сигнал VALUE имеет значение, лежащее между 61 и 70 или между 71 и 75;
- выходной порт A = '1', когда сигнал VALUE имеет значение, лежащее между 86 и 100.

При всех остальных значениях сигнала VALUE выходной порт F = '1'. Проиллюстрируем эти положения на следующем примере:

```
library ieee;
use ieee.std_logic_1164.all;
entity GRD_201 is
   port(VALUE: in integer range 0 to 100;
        A, B, C, D: out  bit);
end GRD_201;
architecture behav_grd of GRD_201 is
begin
  process (VALUE)
 begin
    A <= '0';
    B <= '0';
    C <= '0';
    D <= '0';
    F <= '0';
    case VALUE is
         when 51 to 60 =>
              D <= '1';
         when 61 to 70 | 71 to 75 =>
              C <= '1';
         when 76 to 85 =>
              B <= '1';
         when 86 to 100 =>
              A <= '1';
         when others =>
              F <= '1';
    end case;
```

```
  end process;
end behav_grd;
```

Обратите внимание на операцию, содержащуюся в проекте и обозначенную символом вертикальной черты (|), которая является эквивалентной операции «ИЛИ». Это очень полезная операция, которая позволяет в одном выражении указать несколько несмежных диапазонов (например, для представления диапазона [0...4; 6...10] следует указать в операторе `0 to 4|6 to 10`).

Ниже приведен пример проекта мультиплексора 4×1, который построен с использованием оператора **case**.

```
entity MUX_4_1 is
port (SEL: in std_logic_vector(2 downto 1);
    A, B, C, D: in std_logic;
    Z: out std_logic);
end MUX_4_1;
architecture behav_MUX41 of MUX_4_1 is
begin
PR_MUX: process (SEL, A, B, C, D)
 begin
   case SEL is
        when "00" => Z <= A;
        when "01" => Z <= B;
        when "10" => Z <= C;
        when "11" => Z <= D;
        when others => Z <= 'X';
   end case;
end process PR_MUX;
end behav_MUX41;
```

В этом примере конструкция **when others** определяет блок операторов (в данном случае представлен оператором установки значения сигнала Z) в тех ситуациях, когда задающее выражение SEL имеет значение "0X", "0Z", "XZ", "UX" и т. д.

Примечание. *Данная комбинационная схема может быть описана и другими способами, например с помощью операторов параллельной обработки* ***when ... select***.

8.4. Оператор цикла LOOP

Оператор цикла **loop** используется для многократного выполнения блока последовательных операторов, располагаемых в его теле. Оператор **loop** имеет следующий обобщенный синтаксис:

```
[ метка_цикла :]  итерационная_схема loop
блок_операторов;
   [next [метка_перехода] [when выражение_перехода];]
   [exit [метка_выхода] [when выражение_выхода];]
end loop [метка_цикла];
```

Метки являются необязательными элементами конструкции оператора цикла, но они очень полезны при записи вложенных циклов. На практике оператор цикла **loop** применяется не в форме обобщенного синтаксиса, а в форме базового синтаксиса (без итерационной схемы) либо в форме синтаксиса с одной из двух итерационных схем (**while** ... **loop** и **for** ... **loop**). Далее в этом разделе мы рассмотрим все три формы подробнее.

Операторы вспомогательной передачи управления — **next** и **exit** являются последовательными операторами, которые используются лишь для формирования тела оператора **loop**. Обладают следующими особенностями:

- Оператор **next** приостанавливает в текущей итерации выполнение операторов тела цикла, расположенных под ним, и осуществляет переход на следующую итерацию цикла.
- Оператор **exit** приостанавливает выполнение операторов тела цикла, которые расположены под ним, и передает управление за пределы тела цикла, тем самым полностью прекращая его выполнение.

8.4.1. Базовая форма цикла

Как было сказано выше, оператор цикла **loop** в базовой форме не содержит в своей структуре итерационной схемы. В остальном его синтаксис по своему устройству совпадает с приведенным выше обобщенным синтаксисом.

```
[ метка_цикла :]  loop
блок_операторов;
```

```
   [next [метка_перехода] [when выражение_перехода];]
   [exit [метка_выхода] [when выражение_выхода];]
end loop [метка_цикла];
```

Оператор цикла в базовой форме запускается на исполнение без анализа каких-либо условий и выполняется непрерывно, пока в его теле не встретится оператор **exit** или **next**. Такой оператор **loop** (так же, как и оператор цикла в форме с итерационной схемой **while** … **loop**) должен иметь в своем теле не менее одного оператора **wait**.

В качестве примера можно привести проект пятибитного счетчика с диапазоном выходных значений от 0 до 31 (*см.* пример 8.1). Когда значение счетчика достигает 31, на следующей итерации отсчет начинается снова от 0. В теле цикла имеется оператор **wait**, чтобы оператор **loop** изменял выходные значения каждый раз при обнаружении переднего фронта тактового импульса CLK, т. е. при изменении значения сигнала CLK из '0' в '1'.

Пример 8.1. Использование базовой формы оператора loop для реализации 5-битного счетчика

```
entity COUNT31 is
port (CLK: in std_logic;
  COUNT: out  integer);
end COUNT31;
architecture behav_COUNT of COUNT31 is
begin
P_COUNT: process
   variable intern_value: integer :=0;
 begin
   COUNT <= intern_value;
 loop
   wait until CLK='1';
     intern_value:=(intern_value + 1) mod 32;
     COUNT <= intern_value;
 end loop;
end process P_COUNT;
end behav_COUNT;
```

Примечание. *В проекте в теле оператора* **process** *дополнительно определена переменная* **intern_value**, *потому что выходные порты не могут считывать в теле процесса.*

8.4.2. Итерационная форма цикла WHILE ... LOOP

Оператор цикла формы **while** ... **loop** перед выполнением каждой итерации проверяет истинность выражения, имеющего тип boolean. Когда выражение имеет значение TRUE, оператор цикла выполняет очередную итерацию, в противном случае выполнение тела цикла либо вообще пропускается, либо приостанавливается. Синтаксис оператора формы **while** ... **loop** выглядит следующим образом:

```
[ метка_цикла :] while выражение loop
 блок_операторов;
   [next [метка_перехода] [when выражение_перехода];]
   [exit [метка_выхода] [when выражение_выхода];]
   [блок_операторов;]
 оператор_управления_переменной_цикла;
end loop [метка_цикла];
```

Выражение, стоящее после ключевого слова **while** в операторе цикла, может быть логико-алгебраическим или логическим и содержать *переменную цикла* (loop variable). Начальное значение переменной цикла должно быть известно до начала выполнения цикла формы **while ... loop**. Важно также помнить, что последним оператором в теле цикла должен быть оператор, изменяющий значение переменной цикла, в противном случае может возникнуть ситуация бесконечного цикла (так называемое зацикливание).

Далее приведен пример, иллюстрирующий правила применения и оформления оператора цикла **loop** в итерационной форме **while** ... **loop**:

```
entity While_loop1 is
 port (inp : in bit_vector(3 downto 0);
    Enab   : in bit;
     Res   : out  bit_vector(3 downto 0));
end While_loop1;
architecture While_loop1_ar of While_loop1 is
begin
 process (inp)
  variable k1 : integer; -- Объявление переменной цикла.
 begin
  K1   := 0;
  while (k1 <= 3) loop
```

```
    Res(k1) <= enab and inp(k1);
    K1 := k1 + 1;
  end loop;
 end Process;
end While_loop1_ar;
```

8.4.3. Итерационная форма цикла FOR ... LOOP

Оператор цикла формы **for** ... **loop** использует целочисленную итеративную схему, в которой заранее задано количество выполняемых итераций. Синтаксис оператора формы **for** ... **loop** выглядит следующим образом:

```
[ метка_цикла :] for переменная_цикла in диапазон loop
 блок_операторов;
   [next [метка_перехода] [when выражение_перехода];]
   [exit [метка_выхода] [when выражение_выхода];]
end loop [метка_цикла];
```

Переменная цикла объявляется автоматически непосредственно в теле цикла **loop**, поэтому ее не нужно объявлять явно вне цикла. Значение переменной цикла в теле цикла может лишь считываться, а за пределами цикла вообще недоступно. В отличие от циклов **while** ... **loop**, в которых работа цикла основана именно на изменении значения переменной цикла, в циклах **for** ... **loop** попытки изменения значения переменной цикла недопустимы. Диапазон должен быть целочисленным диапазоном, заданным в восходящей (*начальное_значение* **to** *конечное_значение*) или в нисходящей (*начальное_значение* **downto** *конечное_значение*) форме.

8.4.4. Операторы NEXT и EXIT

Как было сказано выше, оператор **next** приостанавливает в текущей итерации выполнение операторов тела цикла, расположенных под ним, и осуществляет переход на следующую итерацию цикла. Такой переход реализуется при условии истинности логико-алгебраического *выражения перехода*. Синтаксис оператора **next** имеет вид

```
next [метка_перехода] [when выражение_перехода];
```

Ключевое слово **when** является необязательным. Переход к следующей итерации оператора цикла **loop** реализуется, когда логико-алгебраическое выражение перехода, стоящее после ключевого слова **when**, имеет значение TRUE.

В отличие от оператора **next** оператор **exit** приостанавливает выполнение операторов тела цикла, которые расположены под ним, и передает управление за пределы тела цикла, тем самым полностью прекращая его выполнение. Управление процессом вычислений передается либо на ближайший оператор, стоящий после тела цикла, либо на оператор, помеченный меткой выхода, при условии, что логико-алгебраическое выражение перехода, стоящее после ключевого слова **when**, имеет значение TRUE.

Оператор **exit** имеет следующий синтаксис:

```
exit [метка_выхода] [when выражение_выхода];
```

Ключевое слово **when** является необязательным. Завершение выполнения оператора цикла **loop** прекращается окончательно, когда условие, стоящее после ключевого слова **when**, имеет значение TRUE.

8.5. Оператор ожидания WAIT

Оператор **wait** приостанавливает выполнение тела оператора **process**, до тех пор, пока не происходит очередное событие. В общем случае язык VHDL поддерживает несколько форм оператора **wait**, а именно:

```
wait until условие;
wait for выражение_типа_time;
wait on сигнал;
wait;
```

Однако на практике не все САПР поддерживают все четыре формы оператора **wait**. Примеры оформления оператора **wait** первой формы показаны ниже:

```
wait until сигнал = значение;
wait until сигнал'event and сигнал = значение;
wait until not сигнал'stable and сигнал = значение;
```

Для остановки выполнения оператора **process** условие, заданное в операторе **wait until**, должно иметь значение TRUE. Примеры практического использования таких операторов:

```
wait until CLK='1';
wait until CLK='0';
wait until CLK'event and CLK='1';
```

В первом примере оператор **process** будет приостановлен (т. е. переведен в состояние ожидания), когда фронт сигнала равен 1. Во втором примере оператор **process** будет приостановлен, когда фронт сигнала равен 0. В третьем примере оператор **process** будет приостановлен при обнаружении переднего фронта единичного синхроимпульса.

Внимание! *Оператор* ***process****, содержащий оператор* ***wait****, не может иметь список чувствительности.*

8.6. Оператор NULL

Пустой оператор **null** соответствует состоянию, в котором не выполняется никаких операций. Синтаксис оператора достаточно прост:

```
null;
```

Этот оператор очень полезен в операторе выбора **case**, когда при некотором выборе не следует выполнять какие-либо вычислительные действия.

Ниже приведен пример проекта, где используется пустой оператор. В этом проекте проверяется значение управляющего сигнала CNTL, который может принимать значения от 0 до 31. Когда значение CNTL равно 3 или 15, сигналу Z должно быть установлено значение A **xor** B, а во всех остальных случаях не следует выполнять никакие другие операции, что и делает уместным использование оператора **null**.

```
entity EX_WAIT is
port (CNTL  : in integer range 0 to 31;
    A, B    : in std_logic_vector(7 downto 0);
    Z: out std_logic_vector(7 downto 0));
end EX_WAIT;
architecture arch_wait of EX_WAIT is
```

```
begin
P_WAIT      : process (CNTL)
  begin
   Z <=A;
   case CNTL is
        when 3 | 15 =>
             Z <= A xor B;
        when others =>
             null;
   end case;
end process P_WAIT;
end arch_wait;
```

8.7. Пример VHDL-проекта

Приведенный ниже проект моделирует работу ЦУ, которое вычисляет значения квадратного корня из вещественных чисел, поступающих на входной порт A (диапазон допустимых значений — от 0.0 до 32563.0). Результаты вычислений устанавливаются на выходном порту B. Проект имеет поведенческую форму на основе явно заданного оператора **process**, в котором используются операторы **if** и **while** ... **loop**.

```
entity manja_sqrt is
port (A     : in real range 0.0 to 32563.0;
      B     : out  real range 0.0 to 32563.0);
end manja_sqrt;
architecture manja_sqrt_a of manja_sqrt is
constant eps: real := 0.00001;
begin
 LABEL1     : process (A)   -- Метку LABEL1 можно не задавать.
 -- Объявление переменных.
    variable x  : real;
    variable b1 : real;
 -- Функция SQRT вычисляет квадратный корень аргумента.
 -- В данном случае функция основана на аппроксимации
 -- Ньютона-Рафсона: F(n+1) = 0.5*[F(n) + x/F(n)].
function SQRT (x: real) return real is
constant inival : real := 1.5;
-- Вычисление первых двух переменных ряда.
```

```
variable oldval: real := inival;
variable newval: real := (x/oldval + oldval)/2.0;
begin
-- Проверка корректности аргумента.
 if (x < 0.0) then
 assert false report "X < 0 in SQRT(X)" severity ERROR;
 return(0.0);
 end if;
-- Вычисление остальных слагаемых ряда.
 while (abs(newval – oldval) > eps) loop
  oldval := newval;
  newval := (x/oldval + oldval)/2.0;
 end loop;
 return newval;
end SQRT;
begin
 if A > 0.0 then
  x  := A;
  b1 := SQRT(x);
  B <= b1;
 end if;
end process;
end manja_sqrt_a;
```

Временная диаграмма работы проекта представлена на **Рис. 8.2**.

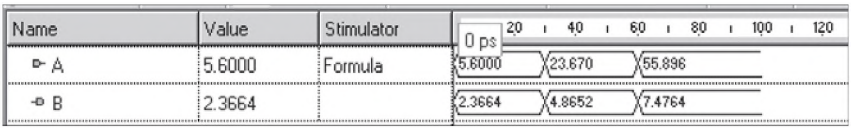

Рис. 8.2. Пример временной диаграммы для моделирования вычисления квадратного корня

ГЛАВА 9

ПОВЕДЕНЧЕСКАЯ ФОРМА ПРОЕКТА: НЕЯВНО ЗАДАННЫЙ ОПЕРАТОР PROCESS

В предыдущей главе было показано, что поведенческая форма VHDL-проекта (как ЦУ регистровой логики, так и ЦУ комбинаторной логики) может быть создана на основе явно заданных операторов **process**, в которых используются блоки операторов последовательной логики. В этой главе мы рассмотрим специальный класс операторов **process**, которые в языке VHDL называются *неявно заданными*. Это означает, что в конструкции *всех* разновидностей этого оператора отсутствует ключевое слово **process**. Заметим, язык VHDL поддерживает три формы синтаксиса неявно заданного оператора **process**. Все эти формы синтаксиса могут использоваться *исключительно* для создания проектов ЦУ комбинаторной логики. Формы синтаксиса неявно заданного оператора **process** получили следующие названия:

- Простая параллельная установка значений сигналов (simple concurrent signal assignments).
- Условная установка значений сигналов (conditional signal assignments).
- Селективная установка значений сигналов (selected signal assignments).

Неявно заданные операторы **process** в любой из этих трех форм являются операторами *параллельной обработки* (concurrent statement). Это означает, что они выполняются параллельно и запускаются на выполнение происходящими на сигналах *событиями* (*event*), т. е. управляются событиями. Иными словами, операторы параллельной обработки запускаются на выполнение, как только происходит событие, содержащееся в том или ином виде в конструкции оператора. Все три формы неявно заданных операторов **process** располагаются в архитектуре проекта в разделе выполняемых операторов параллельной обработки.

В последующих разделах этой главы мы детально рассмотрим синтаксические конструкции операторов параллельной обработки, используемых для создания поведенческих форм проектов ЦУ комбинаторной логики. Как будет показано в гл. 10, подобные конструкции отличаются от конструкций, используемых для создания структурных форм проектов ЦУ, в которых описывается схема в терминах компонентов и их соединений.

9.1. Простая параллельная установка значений сигналов

Оператор параллельной обработки (ОПО), используемый для простой установки значения сигнала (ПУЗС), имеет следующий обобщенный синтаксис:

```
Сигнал <= выражение;
```

Этот оператор устанавливает значение вычисленного *выражения* указанному в операторе *сигналу*. Как видно из синтаксиса, конструкция ОПО ПУЗС состоит из трех частей:

1. Идентификатор сигнала, для которого устанавливается значение вычисленного выражения.
2. Символы операции установки значения сигнала (<=).
3. Вычисляемое выражение оператора, которое может быть по своей структуре алгебраическим, логическим или логико-алгебраическим.

Ниже приведены некоторые примеры оформления операторов ПУЗС с указанием в комментарии типа выражения, используемого в каждом примере.

```
Sum <= (A xor B) xor Cin;              -- Логическое выражение.
Carry <= (A and B);                    -- Логическое выражение.
Z <= (not X) or Y after 2 ns;          -- Логическое выражение.
Y12 <= x1*x3 - x1**2 + integer(34.6); -- Алгебраическое выражение.
Z_55 <= x1/x2 * real(567);             -- Алгебраическое выражение.
Sig46 <= (2*A23) < integer(34.6)-34;   -- Логико-алгебраическое
                                          выражение.
```

```
B52 <=  Sig46 /= 69;                    -- Логико-алгебраическое
                                           выражение.
```

При формировании конструкций ОПО ПУЗС необходимо помнить, что тип вычисленного значения выражения должен совпадать с типом сигнала, для которого устанавливается это значение. Это правило обусловлено тем, что язык VHDL является языком со строгой типизацией данных (*см.* гл. 6).

В примере 9.1 приведен VHDL-проект 4-битного сумматора, в котором используются операторы ПУЗС. Обратите внимание на то, что в этом проекте для поддержки операции арифметического сложения над данными типа `std_logic_vector` к проекту подключается пакет `ieee.std_logic_unsigned`. Символ **&** обозначает операции конкатенации. На **Рис. 9.1** представлена временная диаграмма моделирования работы проекта.

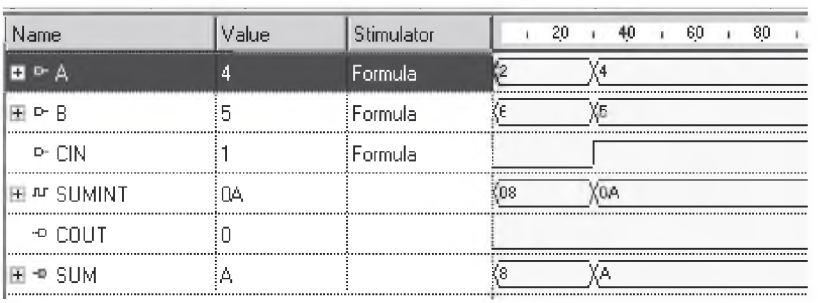

Рис. 9.1. Временная диаграмма моделирования работы проекта 4-битного сумматора (пример 9.1)

Пример 9.1. Проект 4-битного сумматора в поведенческой форме на основе ОПО ПУЗС

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ADD4 is
port (
A: in std_logic_vector (3 downto 0);
B: in std_logic_vector (3 downto 0);
CIN: in std_logic;
```

```
SUM: out std_logic_vector (3 downto 0);
COUT: out std_logic);
end ADD4;
architecture ADD4_concurnt of ADD4 is
-- Объявление внутреннего 5-битного сигнала.
SUMINT : std_logic_vector(4 downto 0);
begin
-- Раздел операторов параллельной обработки архитектуры проекта.
SUMINT <= ('0' & A) + ('0' & B) + ("0000" & CIN);
COUT <= SUMINT(4); -- Перенос в старший бит.
SUM <= SUMINT(3 downto 0); -- Результат суммирования.
end ADD4_concurnt;Обратите внимание на то, что для проверки
работоспособности проекта входные данные задавались в
шестнадцатеричной системе счисления, поэтому на временной
диаграмме работы проекта результаты вычислений также отображены в
шестнадцатеричной системе счисления.
```

9.2. Условная установка значения сигнала

Синтаксис оператора условной установки значения сигнала (УУЗС) имеет следующий вид:

```
Сигнал <= выражение1 when условие1 else
      выражение2 when условие2 else
      ...
      выражениеN;
```

Иными словами, в операторе УУЗС *сигнал* устанавливается равным значению *выражения1*, если *условие1* имеет значение TRUE. В противном случае проверяется *условие2* и т. д. При этом, если ни одно из указанных условий не будет иметь значение TRUE, сигнал будет установлен равным значению *выраженияN*. Если же значение TRUE имеют несколько условий, сигнал устанавливается равным тому значению, условие которого указано в списке первым.

Необходимость введения в язык VHDL оператора УУЗС вызвана тем, что некоторые ЦУ (шифраторы, дешифраторы, мультиплексоры, демультиплексоры), оставаясь по сути комбинаторными устройствами, требуют наличия в своей структуре таких управляющих сигналов, как выбор номера канала, выбор разрядности канала и т. д. В примере 9.2

приведен VHDL-проект 4-битного мультиплексора 4×1, построенного на основе операторов УУЗС.

Пример 9.2. Проект 4-битного мультиплексора 4×1 в поведенческой форме на основе операторов УУЗС

```
entity MUX_4_1_Conc is
port (S1, S0, A, B, C, D: in std_logic;
    Z: out std_logic);
end MUX_4_1_Conc;
architecture concurr_MUX41 of MUX_4_1_Conc is
begin
   Z <= A when S1='0' and S0='0' else
        B when S1='0' and S0='1' else
        C when S1='1' and S0='0' else
        D;
end concurr_MUX41;
```

Операторы УУЗС всегда отслеживают значения содержащихся в них выражений и изменяют значения сигналов, к которым эти операторы применяются, в соответствии с изменением значений выражений. Можно сказать, что конструкции **when** … **else** оператора УУЗС фактически выражают логическую функцию, описывающую работу комбинаторного устройства в форме таблицы истинности.

Проект, представленный в примере 9.2, можно записать в более компактном виде с использованием битовых строк:

```
entity MUX_4_1_funcTab is
port (A, B, C, D: in std_logic;
    SEL: in std_logic_vector (1 downto 0);
    Z: out std_logic);
end MUX_4_1_ funcTab;
architecture concurr_MUX41 of MUX_4_1_ funcTab is
 begin
    Z <= A when SEL = "00" else
         B when SEL = "01" else
         C when SEL = "10" else
         D;
end concurr_MUX41;
```

Обратите внимание на то, что эта конструкция более проста, чем конструкции **if** … **then** … **else** или оператор **case**, используемые

в явно заданном операторе **process**, в примерах проектов мультиплексоров, рассмотренных в разд. 8.2 и 8.3.

9.3. Селективная установка значений сигналов

Оператор селективной установки значения сигнала (СУЗС) подобен описанному в предыдущем разделе оператору УУЗС. Это объясняется тем, что по функциональному назначению они практически идентичны и решают задачу выбора требуемого в текущий момент направления приема или передачи потока информации в ЦУ.

Синтаксис оператора СУЗС имеет следующий вид:

```
with задающее_выражение select
 сигнал <= выражение1 when значение1,
           выражение2 when значение2,
           ...
          [выражениеN when others];
```

Иными словами, в операторе СУЗС *сигнал* устанавливается равным значению того выражения, номер которого определяется значением *задающего выражения*. Поэтому задающее выражение всегда анализируется компилятором на предмет его тождественного равенства значению одного из возможных вариантов выбора. Последние могут быть как выражениями статического типа (например, числом `5`), так и диапазоном (например `4 to 9`) либо комбинацией значений, задаваемых с помощью символа операции ИЛИ (`|`), например:

```
with selector select
 target <= value1 when "000",
           value2 when "001" | "011" | "101",
           value3 when others;
```

Разработчик при формировании значений вариантов выбора должен соблюдать следующие требования.

- В операторе СУЗС не должно быть двух одинаковых значений вариантов выбора.
- Если в операторе СУЗС отсутствует конструкция **when others**, среди значений вариантов выбора, перечисленных в операторе, должны присутствовать *все* возможные значения задающего выражения.

Подобно операторам УУЗС, операторы СУЗС пригодятся для описания работы устройств, работа которых может быть представлена в форме таблицы истинности (примеры 9.3 и 9.4).

Пример 9.3. Проект 4-битного мультиплексора 4×1 в поведенческой форме на основе операторов СУЗС

```
entity MUX_1_4_Conc2 is
port (A, B, C, D: in std_logic;
    SEL: in std_logic_vector(1 downto 0);
    Z  : out std_logic);
end MUX_1_4_Conc2;
architecture concurr_MUX1_4b of MUX_1_4_Conc2 is
 begin
   with SEL select
                  Z <= A when "00",
                       B when "01",
                       C when "10",
                       D when "11";
end concurr_MUX1_4b;
```

Пример 9.4. Проект полного однобитного сумматора, реализованного в поведенческой форме на основе операторов СУЗС

```
library ieee;
use ieee.std_logic_1164.all;
entity FullAdd_Conc is
port (A, B, C: in std_logic;
    sum, cout: out std_logic);
end FullAdd_Conc;
architecture FullAdd_Conc of FullAdd_Conc is
-- Объявление внутреннего сигнала INS с типом std_logic_vector.
signal INS: std_logic_vector (2 downto 0);
begin
-- Установка значений сигнала INS.
   INS(2) <= A;
   INS(1) <= B;
   INS(0) <= C;
with INS select
   (sum, cout) <=
          std_logic_vector'("00") when "000",
```

```
            std_logic_vector'("10") when "001",
            std_logic_vector'("10") when "010",
            std_logic_vector'("01") when "011",
            std_logic_vector'("10") when "100",
            std_logic_vector'("01") when "101",
            std_logic_vector'("01") when "110",
            std_logic_vector'("11") when "111",
            std_logic_vector'("11") when others;
end FullAdd_Conc;
```

В примере 9.4 объявлен вспомогательный сигнал `INS` как вектор `std_logic_vector`, состоящий из трех элементов. Затем элементам этого вектора устанавливаются значения входных портов, после чего они используются в качестве задающего выражения оператора СУЗС. Сигнал, устанавливаемый в операторе СУЗС, имеет составную конструкцию перечислимого типа `(sum, cout)`. Использование такой конструкции позволяет с помощью одного оператора СУЗС одновременно устанавливать значения двух сигналов — `sum` и `cout`. Кроме того, в СУЗС-выражении использованы атрибуты вида `std_logic_vector'("01")`. Такие атрибуты выполняют преобразование символьной строки (например, строки `"01"`) в ее битовое представление (в форму `01`). Наличие битового представления позволяет установить соответствующие значения сигналов, указанных в перечислении (в данном случае сигналу `sum` устанавливается значение `0`, а сигналу `cout` — значение `1`).

ГЛАВА 10

СТРУКТУРНАЯ ФОРМА ПРОЕКТА

Поскольку в разд. 3.3 методика создания проектов в структурной форме была описана достаточно подробно, мы не будем здесь на ней останавливаться, а лишь подчеркнем, что любой проект ЦУ, имеющий структурную форму, является иерархическим проектом как минимум с двумя уровнями иерархии.

Это объясняется тем, что структурная форма проекта описывает ЦУ в терминах *компонентов* (component) и их связей между собой. При этом предполагается, что каждый компонент уже был объявлен ранее (например, в пакете) и в свою очередь может представлять VHDL-проект более низкого уровня иерархии как в структурной, так и в поведенческой форме. Тем не менее на самом низком уровне иерархии каждый компонент должен представлять собой VHDL-проект поведенческой формы, реализованный с помощью соответствующих операторов (*см.* гл. 8 и 9).

Таким образом, неоспоримым достоинством структурной формы проекта является то, что она позволяет описывать сложные ЦУ на основе заранее определенного ограниченного набора компонентов.

Структурную форму проекта лучше всего сравнивать с изображением традиционной блок-схемы ЦУ, которая состоит из функциональных узлов цифровой схемотехники (компоненты VHDL-проекта) и их соединений (*см.* **Рис. 2.3**).

Язык VHDL обеспечивает формальный способ воплощения такой аналогии с помощью следующих подходов.

- Объявление перечня используемых компонентов.
- Объявление в проекте структурной формы внутренних сигналов, которые определяют связи между компонентами, содержащимися в проектируемом ЦУ. Кроме того, в проекте объявляются сиг-

налы, определяющие связи между внешними портами проекта и компонентами, включенными в проект.

- Реализация множества экземпляров одного и того же компонента, что позволяет рассматривать каждый экземпляр как уникальный программный объект с сохранением общих характеристик, присущих базовому компоненту.

Компоненты и сигналы объявляются в разделе объявлений архитектуры проекта, как показано ниже.

```
architecture имя_архитектуры of ИМЯ_КОМПОНЕНТА is
   -- Объявления.
         объявления_компонентов;
         объявления_сигналов;
   begin
   -- Операторы.
         операторы_создания_экземпляров_компонентов;
...
end имя_архитектуры;
```

10.1. Оператор COMPONENT

Для того чтобы в проекте можно было создать экземпляры компонентов и реализовать связи между ними, эти компоненты должны быть объявлены либо в разделе объявлений архитектуры проекта, либо в пакете, подключенном к проекту. Объявление компонента содержит его идентификатор, а также объявление интерфейса (описание входных и выходных портов). Объявление компонента осуществляется с помощью оператора **component** и имеет синтаксис

```
component имя_компонента [is]
[port (имя_сигнала: mode тип;
   имя_сигнала: mode тип;
   ...
   имя_сигнала: mode тип);]
end component [имя_компонента];
```

Идентификатор объявленного компонента (*имя_компонента*) соответствует аналогичному идентификатору проекта, интерфейс которого определен с помощью оператора **entity** либо в библиотеке или в

пакете библиотечных компонентов, либо непосредственно в VHDL-файле проекта структурной формы.

Имена, режимы и типы портов интерфейса, содержащиеся в операторе **component**, должны соответствовать именам, режимам и типам аналогичных портов, содержащихся в объявлении интерфейса компонента в операторе **entity**.

Ниже приведено несколько примеров объявления компонентов.

```
component OR2
    port  (in1, in2: in std_logic;
           out 1: out std_logic);
end component;
component PROC
    port  (CLK, RST, RW, STP: in std_logic;
           ADDRBUS: out std_logic_vector (31 downto 0);
           DATA: inout integer range 0 to 1024);
end component;
component FULLADDER
    port  (a, b, c: in std_logic;
           sum, carry: out std_logic);
end component;
```

Как отмечалось выше, оператор **component** должен находиться либо в архитектуре того проекта, в котором нужно реализовывать экземпляры компонента, либо в отдельном пакете. В последнем случае повторно объявлять компонент в архитектуре проекта не нужно — для реализации его экземпляров достаточно подключить к проекту соответствующие библиотеку и пакет.

10.2. Оператор PORT MAP

Оператор **port map**, о котором также уже рассказывалось в гл. 3, предназначен для создания необходимого количества экземпляров компонента и реализации междукомпонентых связей в проекте структурной формы. Оператор **port map** может сохраняться на текущем уровне иерархии проекта, в пакете библиотеки, поставляемой разработчиком САПР, или в пакете библиотеки, созданной пользователем.

Оператор реализации компонента имеет следующий синтаксис:

```
имя_экземпляра : имя_компонента port map ([порт1 =>] сигнал1,
     [порт2 =>] сигнал2,... , [портN =>] сигналN);
```

В качестве *имени экземпляра* реализуемого компонента можно взять любой идентификатор, допустимый в языке VHDL. Имя экземпляра используется в программном коде проекта для ссылки на конкретный экземпляр компонента, как в случае с любым другим объектом программы. В качестве *имени компонента* берется идентификатор, который применялся для объявления компонента с помощью оператора **component**. Далее с помощью оператора **port map** реализуются междукомпонентные связи заданного экземпляра компонента. Реализация связей выполняется путем создания *схемы* (map) соединений *портов* и внутренних *сигналов* проекта. Список таких портов и сигналов указывается в круглых скобках оператора **port map**.

Как вы уже знаете из гл. 3, междукомпонентные связи каждого экземпляра компонента можно определять с помощью позиционной (неявной) реализации или с помощью именованной (явной реализации). Допускается также комбинированная реализация, при которой в операторе **port map** используются оба метода одновременно.

При позиционной реализации список оператора **port map** имеет следующий вид:

```
port map (сигнал1, сигнал2,… сигналN);
```

В случае использования синтаксиса позиционной реализации первый порт, указанный в списке портов соответствующего оператора **component**, связывается с *сигналом1* в операторе **port map**, реализующем экземпляр этого компонента. Второй порт, указанный в списке портов оператора **component**, связывается с *сигналом2* и т. д. Примером позиционной реализации является экземпляр U1 компонента NAND2 в приведенном ниже фрагменте проекта структурной формы.

При именованной реализации взаимосвязь портов и сигналов экземпляра реализуемого компонента задается в списке оператора **port map** явно с обозначением связи с помощью символов =>. В уже упоминавшемся фрагменте проекта примером использования такой реализации является оператор **port map**, реализующий экземпляр U2 компонента NAND2. Следует заметить, что при именованной реализации по-

рядок расположения пар *порт* => *сигнал* в списке оператора **port map** не имеет значения.

Наконец, как уже отмечалось, допускается использовать синтаксис комбинированной реализации, что показано на примере экземпляра U3 компонента NAND2.

```
component NAND2
   port (in1, in2: in std_logic;
            out 1: out std_logic);
end component;
   signal int1, int2, int3: std_logic;
architecture struct of EXAMPLE is
   U1: NAND2 port map (A,B,int1);
   U2: NAND2 port map (in2=>C, in2=>D, out 1=>int2);
   U3: NAND3 port map (in1=>int1, int2, Z);
   ...
end architecture struct;
```

Рассмотрим еще три примера 10.1...10.3, иллюстрирующие подходы к созданию VHDL-проектов структурной формы.

В примере 10.1 показан подход, предусматривающий наличие мастер-проекта и подчиненных ему субпроектов. В простейшем случае субпроект может быть один, т. е. при таком подходе VHDL-проект всегда будет состоять как минимум из двух проектов: основного мастер-проекта и одного субпроекта.

В примере 10.1 мастер-проект называется Top и содержит в качестве компонента субпроект с именем Logic. Мастер-проект и субпроекты могут храниться в отдельных файлах. Так, в данном примере мастер-проект сохранен в файле Top_file.vhd, а субпроект — в файле Logic_lower.vhd.

Если все подобные файлы находятся в одном каталоге (обычно САПР по умолчанию считает, что имя каталога совпадает с именем мастер-проекта), САПР автоматически связывает файл субпроекта (logic_lower.vhd) с файлом мастер-проекта (Top_file.vhd). Для этого нужно, чтобы выполнялось только одно условие: субпроекты должны быть откомпилированы заранее, начиная от субпроектов самого низкого уровня. Если это условие не выполняется, соответствующий компонент не будет доступен из субпроекта более высокого уровня или из мастер-проекта.

Пример 10.1. Проект со структурной формой, состоящий из мастер-проекта и субпроекта

Файл `Top_file.vhd`, содержащий VHDL-код мастер-проекта `top`:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
entity top is
port(w_in, x_in, y_in      :IN std_logic;
                clock      :IN std_logic;
                z_out      :OUT std_logic);
end top;
architecture a of top is
component logic -- Объявление компонента logic.
port(a,b,c :IN std_logic;
          x :OUT std_logic);
end component;
signal w_reg, x_reg, y_reg, z_reg :std_logic;
begin
-- Оператор port MAP, реализующий межкомпонентные связи
-- экземпляра low_logic компонента logic.
low_logic : logic port MAP(a => w_reg,
                           b => x_reg,
                           c => y_reg,
                           x => z_reg);
process(clock)
begin
if (clock'event and clock='1') then
    -- Операторы установки значений внутренних сигналов
    -- мастер-проекта в соответствии со значениями на
    -- внешних портах.
 w_reg<=w_in;
 x_reg<=x_in;
 y_reg<=y_in;
 z_out<=z_reg;
end if;
end process;
end a;
```

Файл `Logic_lower.vhd`, содержащий VHDL-код субпроекта `logic`:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
entity logic is
port(a,b,c     :        in std_logic;
         x     :        out std_logic);
end logic;
architecture a of logic is
begin
process (a, b, c)
begin
x<=(a and b) or c;
end process;
END;
```

Теперь разберем более сложный пример, основанный на использовании двух субпроектов в одном мастер-проекте. Предположим, нам требуется построить VHDL-проект ЦУ, структурная схема которого показана на **Рис. 10.1**. Такой VHDL-проект удобно реализовать в структурной форме, поскольку при этом мы можем использовать в качестве компонентов ранее созданные субпроекты ЦУ.

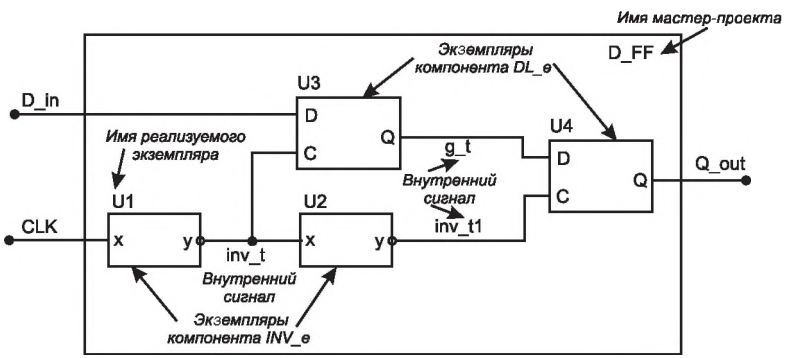

Рис. 10.1. Структурная схема VHDL-проекта ЦУ

Программный код VHDL-проекта представлен в примере 10.2. Мастер-проект `D_FF` использует два субпроекта — `INV_e` и `DL_e`, которые применяются в мастер-проекте в качестве компонентов.

Пример 10.2. Проект со структурной формой, состоящий из мастер-проекта и двух разных субпроектов

Файл D_FF.VHD, содержащий VHDL-код мастер-проекта D_FF:

```
entity D_FF is
    port (D_in : in bit;
          CLK : in bit;
        Q_out : in bit);
end D_FF;
architecture D_FF_a of D_FF is
    component INV_e      -- Оператор объявления компонента INV_e.
          port (X : in bit;
                Y : out  bit);
   end component;
    component DL_e       -- Оператор объявления компонента DL_e.
          port (D, C : in bit;
                   Q : out  bit);
    end component;
    signal q_t, inv_t, inv_t1 : bit;-- Объявление внутренних сигналов.
    begin
        -- Реализация экземпляров U1 и U2 компонента INV_e.
          U1 : INV_e port map (CLK, inv_t);
          U2 : INV_e port map (inv_t, inv_t1);
        -- Реализация экземпляров U3 и U4 компонента DL_e.
          U3 : DL_e port map (D => D_in, C => inv_t, Q => q_t);
          U4 : DL_e port map (D => q_t, C => inv_t1, Q => Q_out);
end  D_FF_a;
```

Файл DL_E.VHD, содержащий VHDL-код субпроекта DL_e:

```
entity DL_e is
    port (D,C : in bit;
           Q : out  bit);
end DL_e;
architecture DL_a of DL_e is
    begin
      U1 : process (C)
           begin
                if C = '1' then Q <= D;
                end if;
           end process;
end DL_a;
```

Файл INV_E.VHD, содержащий VHDL-код субпроекта INV_e:

```
entity INV_e is
   port (X : in bit;
         Y : out  bit);
end INV_e;
architecture INV_a of INV_e is
   begin
        U1: process (C)
            begin
                  Y <= not X;
            end process;
end INV_a;
```

В примере 10.3 мы рассмотрим VHDL-проект 4-битного регистра на базе D-триггера с асинхронным сбросом (**Рис. 10.2**). Его имеет смысл создать в виде проекта со структурной формой, поскольку в этом случае в мастер-проекте можно использовать 4 экземпляра субпроекта D-триггера.

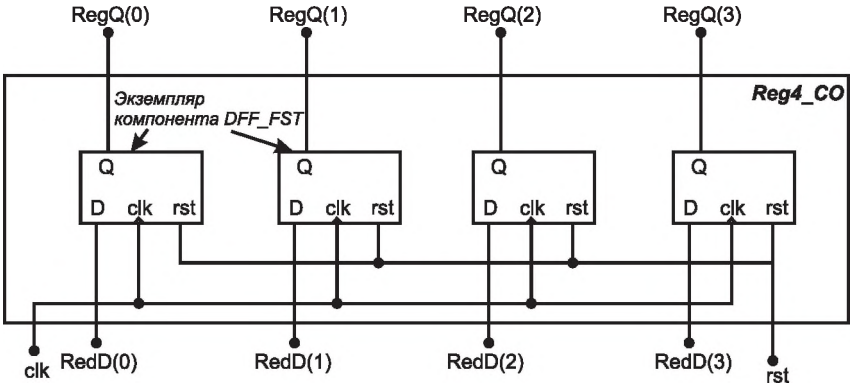

Рис. 10.2. Структурная схема мастер-проекта 4-битного регистра

Пример 10.3. Проект 4-битного регистра со структурной формой, состоящий из мастер-проекта, в котором реализуется 4 экземпляра одного компонента

Файл Reg4_CO.VHD, содержащий VHDL-код мастер-проекта Reg4_CO:

```
entity Reg4_CO is
   port (RedD :  in bit_vector (3 downto 0);
     clk, rst :  in bit;
         RegQ : out  bit_vector (3 downto 0);
end Reg4_CO;
architecture Reg4_CO_arch of Reg4_CO is
   component DFF_RST is           -- Объявление компонента DFF_RST.
   port (D, clk, rst : in bit;
                   Q : out   bit);
   end component;
  begin
-- Реализация четырех экземпляров компонента.
    bit0 :component DFF_RST port map (D => RedD(0),
                        clk => clk, rst => rst, Q => RegQ(0));
    bit1 :component DFF_RST port map (D => RedD(1),
                        clk => clk, rst => rst, Q => RegQ(1));
    bit2 :component DFF_RST port map (D => RedD(2),
                        clk => clk,  rst => rst,  Q => RegQ(2));
    bit3 :component DFF_RST port map (D => RedD(3),
                         clk => clk, rst => rst,  Q => RegQ(3));
end Reg4_CO_arch ;
```

Файл DFF_RST.VHD, содержащий VHDL-код субпроекта D-триггера с асинхронным сбросом DFF_RST :

```
entity DFF_RST is
   port (D, clk,  rst : in bit;
                    Q : out  bit);
end DFF_RST;
architecture DFF_RST_arch of DFF_RST is
 begin
   process (rst, clk)
     begin
          if rst = '1' then  Q <= '0';
          else
              if  clk'event and clk = '1' then
                  Q <= D;
              end if;
         end if;
  end  process;
end  DFF_RST _arch;
```

ГЛАВА 11

ПРИМЕРЫ VHDL-ПРОЕКТОВ ЦУ РАЗЛИЧНОГО НАЗНАЧЕНИЯ

11.1. Проекты ЦУ общего назначения

11.1.1. Логический элемент AND на 4 входа

Работа 4-входового логического элемента `and` описывается таблицей истинности, представленной в **Табл. 11.1**.

Таблица 11.1. **Таблица истинности логического элемента AND на 4 входа**

Вход				Выход
I0	I1	I2	I3	O
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Работа 4-входового элемента **and** может быть также представлена такой булевой функцией:

```
O = I0 and I1 and I2 and I3
```

Ее программная реализация имеет следующий вид:

```
-- ------------------------------------------------
--                    Проект gate_AND
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com ++++++++++++++++
-- ------------------------------------------------
-- Заголовок проекта (раздел подключения библиотек и пакетов).
library ieee; -- Оператор подключения библиотеки.
use ieee.std_logic_1164.all; -- Оператор подключения пакета.
-- Раздел объявлений проекта.
entity gate_and is -- Оператор объявления интерфейса компонента.
   port (      -- Оператор объявления портов интерфейса.
          I0 : in std_logic;
          I1 : in std_logic;
          I2 : in std_logic;
          I3 : in std_logic;
          O  : out std_logic);
end entity;
-- Раздел архитектуры проекта.
architecture gate_arch of gate_and is -- Оператор объявления архитектуры.
-- Раздел объявлений архитектуры проекта.
begin
-- Раздел выполняемых ОПО архитектуры проекта.
O <= I0 and I1 and I2 and I3; -- ОПО ПУЗС.
end architecture gate_arch;
```

Примечание. *В VHDL-проектах значения входных и выходных портов часто имеют тип* `bit`, `boolean`, `std_logic`. *В проекте полужирным шрифтом выделены ключевые слова языка VHDL.*

11.1.2. Логический элемент OR на 2 входа

Работа 2-входового элемента **or** описывается таблицей истинности (**Табл. 11.2**).

Таблица 11.2. **Таблица истинности логического элемента OR на 2 входа**

Вход		Выход
I0	I1	O
0	0	0
0	1	1
1	0	1
1	1	1

Работа 2-входового элемента `or` может быть также представлена булевой функцией:

```
O = I0 or I1
```

Ее программная реализация имеет следующий вид:

```
-- ------------------------------------------------------
--                    Проект gate_OR
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity gate_or is
   port (
          I0 : in std_logic;
          I1 : in std_logic;
          O  : out std_logic
   );
end entity;
architecture gate_arch of gate_or is
begin
   O <= I0 or I1;
end architecture gate_arch;
```

11.1.3. Логический элемент OR на 4 инверсных входа

Работа 4-входового элемента `or` с 4 инверсными входами описывается таблицей истинности (**Табл. 11.3**).

Таблица 11.3. **Таблица истинности логического элемента OR на 4 инверсных входа**

Вход				Выход
I0	I1	I2	I3	O
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Работа 4-входового элемента **or** может быть также представлена булевой функцией:

```
O = not(I0) or not(I1) or not(I2) or not(I3)
```

Ее программная реализация имеет следующий вид:

```
-- ------------------------------------------------------
--                    Проект gate_NOT_OR
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity gate_not_or is
   port (
          I0 : in std_logic;
          I1 : in std_logic;
          I2 : in std_logic;
          I3 : in std_logic;
```

```
          O : out std_logic);
end entity;
architecture gate_arch of gate_not_or is
begin
   O <= not I0 or not I1 or not I2 or not I3;
end architecture gate_arch;
```

11.1.4. Логический элемент NOR на 3 входа

Работа элемента **nor** с 3 входами описывается таблицей истинности (**Табл. 11.4**).

Таблица 11.4. **Таблица истинности элемента NOR на 3 входа**

Вход			Выход
I0	I1	I2	O
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Работа 3-входового элемента **NOR** может быть также представлена булевой функцией:

```
O = not(I0 or I1 or I2)
```

Ее программная реализация имеет следующий вид:

```
-- ------------------------------------------------
--                    Проект gate_NOR
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity gate_nor is
   port (
          I0 : in std_logic;
```

```
          I1 : in std_logic;
          I2 : in std_logic;
          O  : out std_logic);
end entity;
architecture gate_arch of gate_nor is
begin
   O <= not (I0 or I1 or I2);
end architecture gate_arch;
```

11.1.5. Логический элемент XOR на 2 входа

Работа 2-входового элемента **xor** описывается таблицей истинности **(Табл. 11.5)**.

Таблица 11.5. **Таблица истинности элемента XOR на 2 входа**

Вход		Выход
I0	I1	O
0	0	0
0	1	1
1	0	1
1	1	0

Работа 2-входового элемента **xor** может быть также представлена булевой функцией:

```
O = not(I0) and I1 or I0 and not(I1)= I0 xor I1
```

Ее программная реализация имеет следующий вид:

```
-- ------------------------------------------------
--                    Проект gate_XOR
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity gate_xor is
   port (
          I0 : in std_logic;
          I1 : in std_logic;
          O  : out std_logic);
```

```
end entity;
architecture gate_arch of gate_xor is
begin
   O <= I0 xor I1;
end architecture gate_arch;
```

11.1.6. 4-битный буфер — логический усилитель

Работа *одного бита* логического усилителя описывается таблицей истинности (**Табл. 11.6**).

Таблица 11.6. **Таблица истинности одного бита логического усилителя**

Вход	Выход
I	O
0	0
1	1

Работа 4-входового буфера (при условии, что входы и выходы представлены элементами данных типа `std_logic_vector` или `bit_vector`) может быть выражена такой булевой функцией:

```
O = I
```

Ее программная реализация имеет следующий вид:

```
-- ------------------------------------------------------
--                    Проект Buff
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity buff is
   port (
          I : in std_logic_vector(3 downto 0);
          O : out std_logic_vector(3 downto 0)       );
end entity;
architecture buff_arch of buff is
begin
   O <= I;
end architecture;
```

Примечание. *В VHDL-проектах многобитных ЦУ значения входных и выходных портов могут иметь тип как* `bit`, `boolean`, `std_logic`, *так и* `std_logic_vector` *или* `bit_vector`.

11.1.7. 8-битный буфер с тремя состояниями без обратной связи

Этот пример реализует 8-битный буфер с тремя состояниями (tri-state buffer), для чего в архитектуре проекта используется оператор **when ... else**. Этот буфер не имеет обратной связи, поэтому выходной порт my_out объявлен как порт с режимом **out**, а не **inout**:

```
-- ------------------------------------------------------
--                      Проект PreBUS
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity prebus is
port(
my_in  : in std_logic_vector (7 downto 0);
  sel  : in std_logic; -- Входной порт для выбора состояния.
my_out : out std_logic_vector (7 downto 0));
end prebus;
architecture maxpld of prebus is
begin
     -- Оператор УУЗС.
        my_out <= "ZZZZZZZZ" when (sel = '1') else my_in;
end maxpld;
end;
```

11.1.8. 4-канальный мультиплексор 4×1 с сигналом разрешения выбора номера канала

Данный проект, реализующий 4-канальный мультиплексор 4×1 (3 бита в каждом канале), иллюстрирует использование операторов **if ... then** и **case**:

```
-- ------------------------------------------------------
--                      Проект Mux
```

```
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ----------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity mux is
   port (
          EN : in std_logic; -- Сигнал разрешения выбора канала.
          I0 : in std_logic_vector (2 downto 0);
          I1 : in std_logic_vector (2 downto 0);
          I2 : in std_logic_vector (2 downto 0);
          I3 : in std_logic_vector (2 downto 0);
           S : in std_logic_vector (1 downto 0); -- Выбор канала.
           O : out std_logic_vector (2 downto 0));
end entity;
architecture mux_arch of mux is
constant NON_ACTIVE : std_logic_vector (2 downto 0) := (others => '0');
begin
process (S, EN, I0, I1, I2, I3)
  begin
     if (EN = '0') then
       O <= NON_ACTIVE;
     else
       case CONV_INTEGER(S) is
            when 0 => O <= I0;
            when 1 => O <= I1;
            when 2 => O <= I2;
            when 3 => O <= I3;
            when others => O <= NON_ACTIVE;
       end case;
     end if;
end process;
end architecture;
```

11.1.9. Демультиплексор 1×4

```
-- ----------------------------------------------------
--                    Проект DMux
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
```

```
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_UNSIGNED.all;
entity dmux is
   port (
      EN :  in std_logic;      --    Сигнал разрешения выбора канала.
      S  :  in std_logic_vector (1 downto 0);    -- выбор номера канала.
      I  :  in std_logic_vector (7 downto 0);  -- входной канал.
      O0 :  out std_logic_vector (7 downto 0); -- выходной канал 0.
      O1 :  out std_logic_vector (7 downto 0); -- выходной канал 1.
      O2 :  out std_logic_vector (7 downto 0); -- выходной канал 2.
      O3 :  out std_logic_vector (7 downto 0));-- выходной канал 3.
end entity;
architecture dmux_arch of dmux is
constant NON_ACTIVE : std_logic_vector (7 downto 0) := (others => '0');
begin
   O0 <= I when (S = 0) and (EN = '1') else NON_ACTIVE;
   O1 <= I when (S = 1) and (EN = '1') else NON_ACTIVE;
   O2 <= I when (S = 2) and (EN = '1') else NON_ACTIVE;
   O3 <= I when (S = 3) and (EN = '1') else NON_ACTIVE;
end architecture;
```

11.1.10. 4-битный последовательный сумматор с фиксированной точкой

```
-- ------------------------------------------------
--                    Проект Adder
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_UNSIGNED.all;
entity adder is
    port (CI:  in std_logic;   --    Сигнал переноса из младшего бита.
          OV :  out std_logic;  --    Сигнал переполнения.
          CO :  out std_logic;  --    Сигнал переноса в старший бит.
    A, B    : in std_logic_vector (3 downto 0); -- Слагаемые.
          Q  : out std_logic_vector (3 downto 0) -- Сумма.
   );
```

```
end entity;
architecture adder_arch of adder is
begin
    process (A, B, CI)
    variable TEMP_RESULT:std_logic_vector (3 downto 0);
    variable TEMP_RESULT2:std_logic_vector (1 downto 0);
 begin
         TEMP_RESULT:=('0' & A(2 downto 0)) + ('0' & B(2 downto 0)) + CI;
         TEMP_RESULT2:=('0' & A(3)) + ('0' & B(3)) + TEMP_RESULT(3);
         Q <= TEMP_RESULT2(0) & TEMP_RESULT(2 downto 0);
         CO <= TEMP_RESULT2(1);
         OV <= TEMP_RESULT2(1) xor TEMP_RESULT(3);
    end process;
end architecture adder_arch;
```

11.1.11. 8-битный сумматор с ускоренным переносом

Рекурсивное разворачивание переноса во всех битах сумматора в данном проекте реализуется двухуровневым выражением **and ... or**. Специальная обработка в проекте *сигнала переноса* (carry) уменьшает время распространения переноса по битам сумматора, что дает возможность повысить его производительность.

```
-- ------------------------------------------------------
--                      Проект C_L_Addr
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.ALL;
entity c_l_addr is
port(x_in:in std_logic_vector(7 downto 0);
    y_in :  in std_logic_vector(7 downto 0);
carry_in :  in std_logic;
    sum  :  out std_logic_vector(7 downto 0);
carry_out:  out std_logic);
end c_l_addr;
architecture behavioral of c_l_addr is
-- carry_propagate – сигнал распространения переноса.
-- carry_in_internal - входной внутренний сигнал переноса.
signal h_sum             : std_logic_vector(7 downto 0);
```

```
signal carry_generate    : std_logic_vector(7 downto 0);
signal carry_propagate   : std_logic_vector(7 downto 0);
signal carry_in_internal : std_logic_vector(7 downto 1);
begin
h_sum <= x_in xor y_in;
carry_generate <= x_in and y_in;
carry_propagate <= x_in or y_in;
process (carry_generate, carry_propagate, carry_in_internal)
begin
carry_in_internal(1) <= carry_generate(0) or
                        (carry_propagate(0) and carry_in);
inst: for i in 1 to 6 loop
   carry_in_internal(i+1) <= carry_generate(i) or
                        (carry_propagate(i) and
carry_in_internal(i));
      end loop;
carry_out <= carry_generate(7) or
                        (carry_propagate(7) and
carry_in_internal(7));
end process;
sum(0) <= h_sum(0) xor carry_in;
sum(7 downto 1) <= h_sum(7 downto 1) xor carry_in_internal(7
downto 1);
end behavioral;
```

11.1.12. 4-битный каскадный вычитатель с фиксированной точкой

```
-- ------------------------------------------------------
--                    Проект Subtr
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_UNSIGNED.all;
entity subtr is
    port (
          CI : in std_logic;
          OV : out std_logic;
          CO : out std_logic;
```

```
        A, B : in std_logic_vector (3 downto 0);
           Q : out std_logic_vector (3 downto 0)
   );
end entity;
architecture subtr_arch of subtr is
begin
 process (A, B, CI)
 variable TEMP_B : std_logic_vector (3 downto 0);
 variable TEMP_RESULT  : std_logic_vector (3 downto 0);
 variable TEMP_RESULT2 : std_logic_vector (1 downto 0);
    begin
    TEMP_B       := not B;
    TEMP_RESULT   := ('0' & A(2 downto 0)) + ('0' & TEMP_B(2 downto
                                                    0)) + (not CI);
    TEMP_RESULT2 := ('0' & A(3)) + ('0' & TEMP_B(3)) + TEMP_RESULT(3);
    Q <= TEMP_RESULT2(0) & TEMP_RESULT(2 downto 0);
    CO <= not TEMP_RESULT2(1);
    OV <= TEMP_RESULT2(1) xor TEMP_RESULT(3);
 end process;
end architecture subtr_arch;
```

11.1.13. 8-битный сумматор со сквозным переносом с учетом знаков битов

В данном проекте 8-битного сумматора со сквозным переносом показана методика использования оператора **`for ... generate`**, который применяется в архитектуре мастер-проекта (файл `F_Add8.vhd`) для реализации в цикле 6 экземпляров компонентов `full_add` и `carry`. Объявление интерфейса и архитектура субпроекта однобитного полного сумматора `full_add` содержатся в файле проекта `Full_Add.vhd`, который нужно предварительно откомпилировать, прежде чем подключать компонент к `full_add` к мастер-проекту `f_add8`. Объявление интерфейса и архитектура субпроекта `carry` находятся в пакете `maxplus2.carry` библиотеки `altera`, поэтому для реализации экземпляров компонента `carry` соответствующие библиотеку и пакет нужно подключить к мастер-проекту.

Примечание. *Сумматор со сквозным переносом, показанный в этом примере, может использоваться в проектах, где эффективное использо-*

вание ресурсов ПЛИС более важно, чем параметры производительности проекта.

```
-- ------------------------------------------------
--                  Мастер-проект F_Add8
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library altera;
use altera.maxplus2.carry;
library ieee;
use ieee.std_logic_1164.all
entity f_add8 is
port(x_in  :  in std_logic_vector(7 downto 0);
   y_in    :  in std_logic_vector(7 downto 0);
   c_in    :  in std_logic;
    sum    :  out std_logic_vector(7 downto 0);
  c_out    :  out std_logic);
end f_add8;
 architecture struct of f_add8 is
 signal im :  std_logic_vector(6 downto 0);
 signal imi:  std_logic_vector(6 downto 0);
 component full_add
   port(a  :  in std_logic;
        b  :  in std_logic;
     c_in  :  in std_logic;
      sum  :  out std_logic;
    c_out  :  out std_logic);
end component;
begin
c0  : full_add port map (x_in(0),y_in(0),c_in,sum(0),im(0));
c01 : carry port map (im(0),imi(0));
c   : for i in 1 to 6 generate
  c1to6: full_add port map (x_in(i),y_in(i),
                            imi(i-1),sum(i),im(i));
  c11to16: carry port map (im(i),imi(i));
end generate;
c7  : full_add port MAP (x_in(7),y_in(7), imi(6),sum(7),c_out);
end struct;
-- ------------------------------------------------
-- Субпроект Full_Add мастер-проекта F_Add8
```

```
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all
entity full_add is
port( a    :  in std_logic;
    b      :  in std_logic;
 c_in      :  in std_logic;
  sum      :  out std_logic;
c_out      :  out std_logic);
end full_add;
architecture behv of full_add is
begin
   sum <=  a xor b  xor c_in;
 c_out <= (a and b) or (c_in and (a or b));
end behv;
```

11.1.14. Универсальный параллельный арифметический процессор

```
-- ------------------------------------------------------
--                    Проект Arithmetic
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity arithmetic is
port (A, B: in std_logic_vector(3 downto 0) ;-- Входы операндов.
    Q1    : out std_logic_vector(4 downto 0);-- Выход сумматора.
Q2, Q3    : out std_logic_vector(3 downto 0);-- Выход вычитателя и
                                             -- делителя.
    Q4    : out std_logic_vector(7 downto 0));-- Выход умножителя.
end arithmetic;
architecture behav of arithmetic is
begin
    process (A, B)
        begin
```

```
            Q1 <= ('0' & A) + ('0' & B);  -- Сложение.
            Q2 <= A - B;                  -- Вычитание.
            Q3 <= A / B;                  -- Деление.
               Q4 <= A * B; -- Умножение.
   end process;
end behav;
```

11.1.15. 4-битный компаратор с анализом знаков сравниваемых чисел и выбором режимов сравнения

```
-- ------------------------------------------------
--                    Проект Compare
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity compare is
   port (
            inAeB : in   std_logic;
            inAgB : in   std_logic;
            inAmB : in   std_logic;
           outAeB : out std_logic;
           outAgB : out std_logic;
           outAmB : out std_logic;
                A : in   std_logic_vector(3 downto 0);
                B : in   std_logic_vector(3 downto 0));
end entity;
architecture compare_arch of compare is
begin
   outAeB <= '1' when (A = B) and (inAeB = '1') else '0';
   outAgB <= '1' when (SIGNED(A) > SIGNED(B)) or ((SIGNED(A) =
                      SIGNED(B)) and (inAgB = '1')) else '0';
   outAmB <= '1' when (SIGNED(A) < SIGNED(B)) or ((SIGNED(A) =
                      SIGNED(B)) and (inAmB = '1')) else '0';
end architecture compare_arch;
```

11.1.16. 8-битный контроллер четности

В данном примере проиллюстрирован синтаксис объявления и использования функций в языке VHDL.

```
-- ------------------------------------------------------
--                      Проект Parity
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity parity is
   port (
        IE  : in std_logic; -- Выбор режима контроля 1.
        IO  : in std_logic; -- Выбор режима контроля 0.
        OE  : out std_logic;
        OO  : out std_logic;
        D   : in std_logic_vector (7 downto 0)); -- Контролируемый байт.
end entity;
architecture parity_arch of parity is
-- Объявление и определение подпрограммы-функции контроля четности.
function parity_parity_func (INPUT: std_logic_vector(7 downto 0))
   return std_logic is
          variable TEMP: std_logic;
          begin
                 TEMP:= INPUT(INPUT'high);
                 for I in (INPUT'high - 1) downto INPUT'low loop
                          TEMP:= TEMP xor INPUT(I);
                 end loop;
          return TEMP;
end function;
begin
   OE <= parity_parity_func(D) xor IE;
   OO <= parity_parity_func(D) xor IO;
end architecture;
```

11.1.17. Простой D-триггер, переключаемый передним фронтом тактового импульса

```
-- ------------------------------------------------
--                      Проект DFF
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity dff is
    port (data, clk :  in std_logic;
            q       :  out std_logic);
end dff;
architecture behav of dff is
begin
   process (clk) begin
          if (clk'event and clk = '1') then
            q <= data;
          end if;
   end process;
end behav;
```

11.1.18. D-триггер с асинхронным сбросом

Триггер сбрасывается в 0 нулевым значением сигнала `reset` (не ожидая очередного синхроимпульса `clk`) и записывает новые данные `data` по заднему фронту синхроимпульса `clk`.

```
-- ------------------------------------------------
--                   Проект DFF_Async_Rst
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity dff_async_rst is
   port (data, clk, reset : in std_logic;
                        q : out std_logic);
end dff_async_rst;
architecture behav of dff_async_rst is
```

```
begin
   process (clk, reset) begin
          if (reset = '0') then
               q <= '0';
          elsif (clk'event and clk = '0') then
               q <= data;
          end if;
   end process;
end behav;
```

11.1.19. D-триггер с асинхронной предустановкой

Триггер устанавливается в 1 нулевым значением сигнала `preset` (не ожидая очередного синхроимпульса `clk`) и записывает новые данные `data` (изменяет состояние) по переднему фронту синхроимпульса `clk`.

```
-- ------------------------------------------------------
--                  Проект Dff_Async_Pre
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity dff_async_pre is
   port (data, clk, preset : in std_logic;
                         q : out std_logic);
end dff_async_pre;
architecture behav of dff_async_pre is
begin
   process (clk, preset) begin
          if (preset = '0') then
               q <= '1';
          elsif (clk'event and clk = '1') then
               q <= data;
          end if;
   end process;
end behav;
```

11.1.20. D-триггер с асинхронным сбросом и асинхронной предустановкой

Триггер сбрасывается в 0 нулевым значением сигнала `reset` (не ожидая очередного синхроимпульса `clk`), устанавливается в 1 нулевым значением сигнала `preset` (не ожидая очередного синхроимпульса `clk`) и записывает новые данные `data` (изменяет состояние) по переднему фронту синхроимпульса `clk`.

```
-- ------------------------------------------------------
--                    Проект Dff_Async
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity dff_async is
   port (data, clk, reset, preset : in std_logic;
                                q : out std_logic);
end dff_async;
architecture behav of dff_async is
begin
   process (clk, reset, preset)
   begin
          if (reset = '0') then
               q <= '0';
          elsif (preset = '1') then
               q <= '1';
          elsif (clk'event and clk = '1') then
               q <= data;
`              end if;
   end process;
end behav;
```

11.1.21. D-триггер с синхронным сбросом

Триггер сбрасывается в 0 единичным значением сигнала `reset` (при приходе очередного переднего фронта синхроимпульса `clk`) и записывает новые данные `data` (изменяет состояние) также по переднему фронту синхроимпульса `clk`.

```
-- ------------------------------------------------
--                    Проект Dff_Sync_Rst
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- +++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity dff_sync_rst is
   port (data, clk, reset : in std_logic;
                       q : out std_logic);
end dff_sync_rst;
architecture behav of dff_sync_rst is
begin
   process (clk)
      begin
      if (clk'event and clk = '1') then
        if (reset = '1') then
            q <= '0';
         else
            q <= data;
        end if;
      end if;
   end process;
end behav;
```

11.1.22. D-триггер с синхронной предустановкой

Триггер устанавливается в 1 нулевым значением сигнала `preset` (при приходе очередного переднего фронта синхроимпульса `clk`) и записывает новые данные `data` (изменяет состояние) также по переднему фронту синхроимпульса `clk`.

```
-- ------------------------------------------------
--                    Проект Dff_Sync_Pre
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- +++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity dff_sync_pre is
port (data, clk, preset : in std_logic;
```

```
            q  :  out std_logic);
end dff_sync_pre;
architecture behav of dff_sync_pre is
begin
   process (clk)
   begin
        if (clk'event and clk = '1') then
          if (preset = '0') then
              q <= '1';
          else
              q <= data;
          end if;
        end if;
   end process;
end behav;
```

11.1.23. D-триггер с синхронным сбросом и сигналом, разрешающим приход очередного тактового импульса

Триггер сбрасывается в 0 единичным значением сигнала `reset` (при приходе очередного переднего фронта синхроимпульса `clk`) и записывает новые данные `data` (изменяет состояние) также по переднему фронту синхроимпульса `clk`, но только тогда, когда значение разрешающего сигнала `en` равно 1.

```
-- ------------------------------------------------------
--                    Проект Dff_Clk_En
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity dff_clk_en is
   port (data, clk, reset, en : in std_logic;
                           q : out std_logic);
end dff_clk_en;
architecture behav of dff_clk_en is
begin
   process (clk, reset) begin
          if (reset = '0') then
```

```
                q <= '0';
          elsif (clk'event and clk = '1') then
               if (en = '1') then
                 q <= data;
               end if;
          end if;
   end process;
end behav;
```

11.1.24. D-триггер-защелка

Триггер-защелка записывает данные data только когда сигнал enable равен 1.

```
-- ------------------------------------------------------
--                      Проект D_Latch
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity d_latch is
   port(enable, data: in std_logic;
                   y : out std_logic);
end d_latch;
architecture behave of d_latch is
begin
   process (enable, data)
   begin
        if (enable = '1') then
          y <= data;
        end if;
   end process;
end behave;
```

11.1.25. D-триггер-защелка со стробирующим сигналом

Триггер-защелка записывает данные, только если сигнал enable равен 1. Конечное состояние триггера-защелки определяется как значением сигнала data, так и значением стробирующего сигнала gate.

```
-- ------------------------------------------------
--                      Проект D_Latch_E
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity d_latch_e is
port (enable, gate, data : in std_logic;
                      q : out std_logic);
end d_latch_e;
architecture behave of d_latch_e is
begin
   process (enable, gate, data)
   begin
        if (enable = '1') then
          q <= data and gate;
        end if;
   end process;
end behave;
```

11.1.26. 8-битный накапливающий счетчик со счетным входом и асинхронным сбросом

Счетчик подсчитывает количество поступивших на него тактовых импульсов `clk`, когда на его счетном входе `en` присутствует сигнал 1. Счетчик сбрасывается в 0 сигналом `rst`, не ожидая прихода очередного тактового сигнала `clk`.

```
-- ------------------------------------------------
--                      Проект Counter8
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity counter8 is
   port (clk, en, rst : in std_logic;
```

```
                count : out std_logic_vector (7 downto 0));
end counter8;
architecture behav of counter8 is
signal cnt: std_logic_vector (7 downto 0);
begin
   process (clk, en, cnt, rst)
   begin
        if (rst = '0') then
               cnt <= (others => '0');
        elsif (clk'event and clk = '1') then
             if (en = '1') then
               cnt <= cnt + '1';
             end if;
        end if;
   end process;
   count <= cnt;
end behav;
```

11.1.27. 8-битный накапливающий счетчик с синхронной загрузкой и асинхронным сбросом

Счетчик подсчитывает количество поступивших на него тактовых импульсов `clk`, когда на его входе `load` присутствует сигнал 0. При приходе очередного переднего тактового импульса `clk` счетчик устанавливается в состояние `data` (записывает значение `data`), когда на его входе `load` присутствует сигнал 0. Счетчик сбрасывается в 0 сигналом `reset`, не ожидая прихода очередного тактового сигнала `clk`.

```
-- ------------------------------------------------------
--                      Проект Counter
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity counter is
   port (clk, reset, load: in std_logic;
```

```
                    data: in std_logic_vector (7 downto 0);
                   count: out std_logic_vector (7 downto 0));
end counter;
architecture behave of counter is
signal count_i : std_logic_vector (7 downto 0);
begin
   process (clk, reset)
   begin
        if (reset = '0') then
                count_i <= (others => '0');
        elsif (clk'event and clk = '1') then
             if load = '1' then
                 count_i <= data;
             else
                 count_i <= count_i + '1';
             end if;
        end if;
   end process;
   count <= count_i;
end behave;
```

11.1.28. 8-битный накапливающий/вычитающий счетчик с выбором направления счета

```
-- ------------------------------------------------------
--                      Проект Counters
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
entity counters is
port(      d : in integer range 0 to 255;
         clk : in bit;
       clear : in bit;
        load : in bit;
     up_down : in bit;
          qd : out integer range 0 to 255);
end counters;
architecture a of counters is
begin
process (clk)
 variable cnt : integer range 0 to 255;
```

```
 variable direction : integer;
begin
if (up_down = '1') then -- Выбор направления.
    direction := 1;
else
    direction := -1;
end if;
if (clk'EVENT and clk = '1') then
  if (load = '1') then
       cnt := d;
  end if;
  if (clk'EVENT and clk = '1') then
    if (up_down = '1') then
       cnt := cnt + direction;
    else
       cnt := cnt + direction;
    end if;
    -- Организация синхронного сброса счетчика.
    if (clear = '0') then
       cnt := 0;
    end if;
  end if;
end if;
  qd <= cnt;
end process;
end a;
```

11.1.29. N-битный накапливающий счетчик со счетным входом, синхронной загрузкой и асинхронным сбросом

Счетчик сбрасывается в 0 сигналом `rst`, равным 1, не ожидая прихода очередного тактового сигнала `clk`. Счетчик подсчитывает количество поступивших на него тактовых импульсов `clk`, когда на его счетном входе `en` присутствует сигнал 1. При приходе очередного переднего тактового импульса `clk` счетчик устанавливается в состояние `data` (записывает значение `data`), когда на его входе `load` присутствует сигнал 1. Разрядность счетчика (`width`) указывается в операторе **`generic()`**.

```
-- ------------------------------------------------------
--                      Проект Counter
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity counter is
generic (width : integer := 16);
   port (data  : in std_logic_vector (width-1 downto 0);
         load, en, clk, rst : in std_logic;
         q : out std_logic_vector (width-1 downto 0));
end counter;
architecture behave of counter is
signal count : std_logic_vector (width-1 downto 0);
begin
   process(clk, rst)
   begin
          if rst = '1' then
                    count <= (others => '0');
          elsif (clk'event and clk = '1') then
               if load = '1' then
                    count <= data;
               elsif en = '1' then
                    count <= count + 1;
               end if;
          end if;
   end process;
   q <= count;
end behave;
```

11.1.30. Синхронный 4-битный циклический сдвиговый регистр вправо с параллельным входом и выходом

Этот регистр выполняет операцию циклического сдвига вправо только при наличии (обнаружении) очередного переднего фронта тактового сигнала `clk` и значения разрешающего сигнала `ce`, равного 1.

Обнаружение очередного переднего фронта тактового сигнала `clk` осуществляется в проекте с помощью специальной подпрограммы-функции `rising_edge(clk)` (объявление этой функции содержится в ядре языка VHDL). Очистка регистра сигналом `clr`, предустановка сигналом `set` и загрузка данными `data` сигналом `load` также возможна при наличии (обнаружении) очередного переднего фронта тактового сигнала `clk` и значения разрешающего сигнала `ce`, равного 1. Сигналы `clr`, `set` и `load` должны быть единичными при выборе соответствующей операции.

```
-- ------------------------------------------------------
--                    Проект Shft_Reg_ROR
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- +++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity shft_reg_ror is
   port ( CLK : in std_logic;
          CLR : in std_logic;
          SET : in std_logic;
           CE : in std_logic;
         LOAD : in std_logic;
         DATA : in std_logic_vector(3 downto 0);
            O : out std_logic_vector(3 downto 0));
end entity;
architecture shft_reg_arch of shft_reg_ror is
signal TEMP_O : std_logic_vector(3 downto 0);
signal SI : in std_logic;
begin
 process(CLK)
     begin
          if rising_edge(CLK) then
            if CE = '1' then
              if CLR = '1' then
                   TEMP_O <= "0000";
              elsif SET = '1' then
                   TEMP_O <= "1111";
              elsif LOAD = '1' then
                   TEMP_O <= DATA;
              else
```

```
                    SI <= TEMP_O(0);
                    TEMP_O <= SI & TEMP_O(3 downto 1);
                end if;
              end if;
            end if;
 end process;
 O <= TEMP_O;
end architecture;
```

11.1.31. Универсальный синхронный 4-битный сдвиговый регистр вправо с параллельным входом и выходом

В отличие от приведенного в подразд. 11.1.24 этот проект содержит один дополнительный вход (входной порт `SI`), на который может быть подан сигнал, поступающий от другого источника. По очередному переднему фронту тактового сигнала `clk` к значению соответствующего бита `SI` посредством операции конкатенации `&` подсоединяются 3 старших бита, хранящихся в регистре. Младший бит, хранящийся в таком регистре до момента выполнения операции сдвига, теряется.

```
-- ------------------------------------------------------
--                      Проект Shft_Reg
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity shft_reg is
   port ( CLK : in std_logic;
          CLR : in std_logic;
          SET : in std_logic;
           CE : in std_logic;
           SI : in std_logic;
         LOAD : in std_logic;
         DATA : in std_logic_vector(3 downto 0);
            O : out std_logic_vector(3 downto 0));
end entity;
architecture shft_reg_arch of shft_reg is
signal TEMP_O : std_logic_vector(3 downto 0);
```

```
begin
   process(CLK)
    begin
          if rising_edge(CLK) then
            if CE = '1' then
              if CLR = '1' then
                   TEMP_O <= "0000";
              elsif SET = '1' then
                   TEMP_O <= "1111";
              elsif LOAD = '1' then
                   TEMP_O <= DATA;
              else
                   TEMP_O <= SI & TEMP_O(3 downto 1);
              end if;
            end if;
          end if;
   end process;
   O <= TEMP_O;
end architecture;
```

11.1.32. Расширитель разрядности шины знаковыми разрядами

В пакете `std_logic_arith` библиотеки `ieee` определены специальные функции — `SXT()` и `EXT()`, которые можно использовать соответственно для расширения знаком и дополнения нулями сигналов типа `std_logic_vector`. Однако эти функции напрямую не поддерживаются урезанной версией VHDL, которая используется в некоторых САПР. Выход заключается в определении этих функций как VHDL-проектов с последующим подключением этих VHDL-проектов в новые проекты в виде компонентов.

В данном подразделе рассмотрен проект `Sign_Ext.vhd`, предназначенный для расширения шины знаковыми разрядами. А в подразд. 11.1.33 приведен проект `Zero_Ext.vhd`, предназначенный для расширения шины нулями.

```
-- ------------------------------------------------------
--                      Проект Sign_Ext
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
```

```
-- ---------------------------------------------------
library ieee;
use ieee.std_logic_1164.all
use ieee.std_logic_arith.all
entity sign_ext is
-- Оператор generic используется, чтобы определить разрядность
-- (ширину) входных и выходных портов (векторов).
   generic(WIDTH_IN : INTEGER := 4; SIZE_OUT : INTEGER := 8);
   port( d_in : in std_logic_vector(WIDTH_IN-1 downto 0);
         d_out : out std_logic_vector(SIZE_OUT-1 downto 0));
end sign_ext;
architecture behavior of sign_ext is
begin
   process(d_in)
          variable zero: boolean;
       begin
            if (d_in(WIDTH_IN-1) = '0') then
                zero := true;
            else
                zero := false;
            end if;
            for j in WIDTH_IN to (size_out-1) loop
               if (zero) then
                   d_out(j) <= '0';
               else
                   d_out(j) <= '1';
               end if;
             end loop;
             d_out(WIDTH_IN-1 downto 0) <= d_in(WIDTH_IN-1 downto 0);
   end process;
end behavior;
```

Субпроект `Sign_Ext.vhd` можно использовать в проектах более высокого уровня в виде компонента `sign_ext`:

```
component sign_ext
GENERIC (WIDTH_IN: INTEGER; SIZE_OUT: INTEGER);
port ( d_in: in std_logic_vector(WIDTH_IN-1 downto 0);
   d_out: out std_logic_vector(SIZE_OUT-1 downto 0));
end component;
```

11.1.33. VHDL-проект расширителя разрядности шины нулями

```
-- ------------------------------------------------
--                      Проект Zero_Ext
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity zero_ext is
   GENERIC(WIDTH_IN : INTEGER := 4; SIZE_OUT : INTEGER := 8);
   port( d_in : IN  std_logic_vector(WIDTH_IN-1 downto 0);
         d_out : OUT std_logic_vector(SIZE_OUT-1 downto 0));
end ZERO_EXT;
architecture behavior of zero_ext is
begin
process(d_in)
     begin
          for j in width_in to size_out-1 loop
          d_out(j) <= '0';
          end loop;
          d_out(WIDTH_IN-1 downto 0) <= d_in(WIDTH_IN-1 downto 0);
end process;
end behavior;
```

Субпроект `Zero_Ext.vhd` можно использовать в проектах более высокого уровня в виде компонента `zero_ext`:

```
component zero_ext
GENERIC (WIDTH_IN: INTEGER; SIZE_OUT: INTEGER);
port ( d_in: in std_logic_vector(WIDTH_IN-1 downto 0);
      d_out: out std_logic_vector(SIZE_OUT-1 downto 0));
end component;
```

11.1.34. 8-битная двунаправленная шина с тремя состояниями

```
-- ------------------------------------------------
--                        Проект BiDir
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com ++++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all
entity bidir is
port(bidir : inout std_logic_vector (7 downto 0);
   oe, clk : in std_logic;
       inp : in std_logic_vector (7 downto 0);
      outp : out std_logic_vector (7 downto 0));
end bidir;
architecture maxpld of bidir is
-- Внутренний сигнал, сохраняющий значения,
-- принимаемые с входного порта inp.
signal a : std_logic_vector (7 downto 0);
-- Внутренний сигнал, сохраняющий значения,
-- передаваемые на выходной порт outp.
signal b : std_logic_vector (7 downto 0);
begin
process(clk)
       begin
           if clk = '1' and clk'EVENT then
                a <= inp;
             outp <= b;
           end if;
end process;
process (oe, bidir)
       begin
           if (oe = '0') then
               bidir <= "ZZZZZZZZ";
                   b <= bidir;
            else
               bidir <= a;
                   b <= bidir;
            end if;
end process;
end maxpld;
```

11.1.35. Процессор возведения в степень N

```
-- ------------------------------------------------
--                      Проект Ste1_uN
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ste1_un is
    port(clk : in std_logic;
         pok : in ingeger range 0 to 255;
           x : in unsigned (63 downto 0);
           y : out  unsigned (63 downto 0));
end ste1_un;
architecture ste1_un of ste1_un is
begin
   process(clk)
      variable poka  : integer range 0 to 255;
      variable res   : unsigned (63 downto 0);
      variable res1  : unsigned (63 downto 0) :=
X"0000000000000001";
    begin
      if clk'event and clk = '1' then
            res1 :=x; -- X"0000000000000003";
            poka := pok;
            for i in 1 to poka loop
               res := resize((res * res1), 64) ;
            end loop;
            y <= res;
            res := X"0000000000000001";
      end if;
   end process;
end ste1_un;
```

11.1.36. Процессор нахождения факториала

```
-- ------------------------------------------------
--                      Проект Fact
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
```

```
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity fact is
    port(   num : in integer range 1 to 4;
         result : out integer range 1 to 24 := 16#1#);
end fact;
architecture fact_a of fact is
pure function facto(constant N : in integer range 1 to 4) return
                                                        integer is
   variable S:integer;
   begin
          if (N=1) then return 1;
          end if;
          if (N >1) or (N<5) then
            for N in 1 to 4 loop
               S:=facto(N-1)* N;
            end loop;
          end if;
          return S;
end facto;
begin
   result <= facto(num);
end fact_a;
```

11.1.37. Процессор нахождения натурального логарифма

```
-- ------------------------------------------------------
--                    Проект YSK_Ln_Syn
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_signed.all;
entity ysk_ln_syn is
port (A : in real range 0.0 to 32563.0;
```

```
    B : out  real range 0.0 to 32563.0);
end ysk_ln_syn;
architecture ysk_ln_syn_a of ysk_ln_syn is
constant eps : real := 0.00001;
begin
process (A)
variable x   : real;
variable b1  : real;
-- Функция LN() вычисляет натуральный логарифм
-- аргумента с помощью ряда Тейлора:
-- ln(x) = (x-1)-(x-1)**2/2 + (x-1)**3/3 -(x-1)**4/4+ ...; abs(x-1) < 1
function LN (x: real) return real is
-- Вычисление двух первых членов ряда.
variable xlocal: real := x – 1.0;
variable oldval: real := xlocal;
variable num: real := xlocal*xlocal;
variable count : integer := 2;
variable newval: real := oldval – (num/real(count));
variable sign  : integer := 1;
begin
-- Проверка корректности аргумента.


if (x <= 0.0 or x > 2.0) then
assert false report "X <= 0 or X > 2.0 in LN(X)" severity WARNING;
return(-999999999.0);
end if;
-- Вычисление остальных членов ряда.
while (abs(newval – oldval) > eps) loop
   oldval := newval;
      num := num*xlocal;
    count := count +1;
   newval := oldval + real(sign)*num/real(count);
     sign := -1 * sign;
end loop;
return newval;
end LN;
begin -- Тело оператора process.
if A /= 0.0 then
 x  := A;
 b1 := LN(x);
 B <= b1;
```

```
end if;
end process;
end ysk_ln_syn_a;
```

11.1.38. Процессор нахождения квадратного корня

```
-- ------------------------------------------------
--                      Проект YSK_Sqrt
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
entity ysk_sqrt is
port (A : in real range 0.0 to 32563.0;
    B   : out  real range 0.0 to 32563.0);
end ysk_sqrt;
architecture ysk_sqrt_a of ysk_sqrt is
constant eps : real := 0.00001;
begin
process (A)
variable x  : real;
variable b1 : real;
-- Функция SQRT() вычисляет значение квадратного
-- корня аргумента с помощью аппроксимации по
-- методу Ньютона-Рафсона:
-- F(n+1) = 0.5*[F(n) + x/F(n)]
function SQRT (x: real) return real is
constant inival: real := 1.5;
-- Вычисление первых двух членов ряда.
variable oldval: real := inival;
variable newval: real := (x/oldval + oldval)/2.0;
begin
-- Проверка корректности аргумента.
     if (x < 0.0) then
       assert false report "X < 0 in SQRT(X)" severity ERROR;
       return(0.0);
     end if;
-- Вычисление остальных членов ряда.
while (abs(newval – oldval) > eps) loop
   oldval := newval;
   newval := (x/oldval + oldval)/2.0;
```

```
end loop;
return newval;
end SQRT;
begin -- Тело оператора process.
 if A > 0.0 then
 x  := A;
 b1 := SQRT(x);
 B <= b1;
 end if;
end process;
end ysk_sqrt_a;
```

11.1.39. Процессор, конвертирующий 16-ричные значения в данные типа std_logic_vector

Данный проект выполняет конвертирование шестнадцатеричных значений в значения типа `std_logic_vector`. Некоторые стандартные процедуры преобразования типов, определенные в VHDL '87 (IEEE Std 1076-1987) и VHDL '93 (IEEE Std 1076-1993), описаны в разд. 6.3.

```
-- ------------------------------------------------
--                    Проект YSK_Hex2SLV
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all
use ieee.std_logic_arith.all
entity ysk_hex2slv is
port(D : out std_logic_vector(15 downto 0));
end ysk_hex2slv;
architecture a of ysk_hex2slv is
begin
D(7 downto 0) <= to_stdlogicvector(x"FC46");
-- В VHDL'93 для преобразования достаточно
-- убрать комментарий в начале следующей строки,
-- поскольку преобразование типов выполняется в VHDL'93 в неявном виде.
-- D <= x"FC46";
end a;
```

11.2. Проекты ЦУ специального назначения

11.2.1. Генератор тактовых импульсов с периодом следования 100 нс и скважностью 50%

```
-- ------------------------------------------------
--                      Проект Oscill
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity oscill is
   port (CLOCK: out std_logic);
end entity;
architecture oscill_arch of oscill is
begin
process
begin
    CLOCK <= '0';
    wait for 50 ns;
    CLOCK <= '1';
    wait for 50 ns;
end process;
end architecture oscill_arch;
```

11.2.2. Комбинированный генератор псевдослучайных чисел с выходными значениями разных типов

```
-- ------------------------------------------------
--                      Проект Random5
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library synopsys;
use synopsys.distributions.all;
library work;
```

```
use work.converters.all;
entity random5 is
   port(            clk : in  bit;
                res_our : out integer;
             res_out_ch : out character;
         res_out1_std_lv : out std_logic_vector (7 downto 0);
           rand_int1_out : out integer;
                 rand_x : out real := 0.3);
end random5;
architecture random5a of random5 is
begin
   process(clk)
          variable x_out : real;
          variable seed_var : real := 0.7;

          constant A1 : real := 0.4;
          constant B1 : real := 0.7;

          variable rand_int1 : integer ;
          variable rand_int : integer;
        variable res_out_std_lv      : std_logic_vector(7 downto 0);
          variable res_out_ch1, res_out_ch2 :       character;
   begin
          if clk'event and clk = '1' then
            UNIFORM(seed_var, A1, B1, x_out);
            rand_x <= x_out;
            rand_int := integer(x_out* 100.0);
            res_our <= rand_int;
            res_out_ch1 := character'VAL(rand_int);
            res_out_ch <= character'VAL(rand_int);
           res_out_ch2 := res_out_ch1;
            rand_int1 :=character'POS(res_out_ch2);
            res_out_std_lv := conv8(rand_int);
            res_out1_std_lv       <=        res_out_std_lv;
            rand_int1_out <= rand_int1;
          end if;
   end process;
end architecture random5a;
```

11.2.3. Многофункциональный конвертер типов данных

В этом проекте выполняется преобразование данных на основе функций, определенных в пакетах `ieee.numeric_std` и `work.converters`. Так как выходные данные одного и того же типа (например, `daout_std_lv`), пересылаемые на один и тот же порт, могут быть получены с помощью разных конвертирующих функций, определенных в разных библиотеках и пакетах, то для поддержания работоспособности проекта в программном коде некоторые операторы установки значения сигналам `daout...` закомментированы.

```
-- ------------------------------------------------------
--                      Проект Char_Std1
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.NUMERIC_STD.all;
library       WORK;
use WORK.CONVERTERS.all;
entity char_std1 is
    port(         clk : in  std_logic;
                dain_bv : in  bit_vector (7 downto 0);
                dain_ch : in  character;
            dain_std_lv : in  std_logic_vector(7 downto 0);
               dain_int : in  integer range 0 to 255;
               dain_uns : in  unsigned (7 downto 0);
              daout_int : out integer range 0 to 255;
               daout_ch : out character;
              daout_uns : out unsigned (7 downto 0);
               daout_bv : out bit_vector (7 downto 0);
          daout_std_lv : out std_logic_vector(7 downto 0));
end char_std1;
architecture char_std1_ar of char_std1 is
begin
 process(clk)
 begin
      if clk'event and clk = '1' then
          daout_std_lv <= std_logic_vector(dain_uns);
            daout_uns <= unsigned(dain_std_lv);
```

```
            daout_int <= to_Integer(dain_uns);
         -- daout_int <= integer(2.8* real(376)/real(56));
            daout_ch  <= To_character (dain_int);
         -- daout_std_lv <= To_StdLogicVector(dain_bv);
         -- daout_uns <= to_Unsigned(dain_int, 8);
         -- daout_int <= CHARACTER'POS(dain_ch);
         -- daout_ch  <= CHARACTER'VAL(dain_int);
         -- daout_ch  <= CONV(dain_std_lv);
         -- daout_std_lv <= CONV(dain_ch);
            daout_bv <= CONV(dain_std_lv);
         -- daout_std_lv <= CONV8(dain_int);
         -- daout_std_lv <= CONV(dain_bv);
      end if;
 end process;
end char_std1_ar;
```

11.2.4. Целочисленный мультипликативный криптопроцессор

Данный проект реализован на базе операций умножения (деления) и оперирует с исходными данными типа `integer`.

```
-- ------------------------------------------------------
--                      Проект Kod_Dec
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity kod_dec is
      port( clk:  in  std_logic;
            data  :  in  integer range 0 to 255;
            kluch :  in  integer range 0 to 255;
            kod   :  out integer range 0 to 32563;
            dec   :  out integer range 0 to 255;
      );
end kod_dec;
architecture kod_dec_ar of kod_dec is
begin
process(clk)
      variable prom : integer range 0 to 32563;
```

```
begin
      if clk'event and clk='1' then
      prom := data * kluch;-- Шифрование данных с помощью ключа.
      kod <=prom;-- Зашифрованные данные передаем на выходной порт.
      dec <= prom/kluch;-- Дешифрование данных и их пересылка на
                        -- выходной порт.
      end if;
end process;
end kod_dec_ar;
```

11.2.5. Символьный аддитивный криптопроцессор

Данный проект реализован на базе операций сложения (вычитания) и оперирует с исходными данными типа `character`. Поскольку прямые арифметические операции над данными такого типа запрещены, решить проблему корректной работы криптопроцессора можно с помощью функций преобразования типа. С целью обеспечения переносимости криптопроцессора между разными САПР в программном коде не используются библиотечные функции преобразования. Вместо этого все необходимые функции определены разработчиком в разделе объявлений оператора **process**`(clk)`. Особенностью этих функций является то, что многие из них имеют одно и то же имя `CONV()`, поскольку являются *перегружаемыми* (overloaded). Это означает, что при моделировании работы проекта VHDL-компилятор САПР автоматически выбирает нужную функцию на основе анализа типа аргумента и возвращаемого результата.

```
-- ------------------------------------------------
--                  Проект Sim_To_Conv_Pack
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity sim_to_conv_pack is
      port(    clk    :  in    std_logic;
         textsim     :  in    character;
         texkod      :  in    character;
         shifrkod    :  out   integer range 0 to 64567;
         deshirtext  :  out   character);
```

```
end sim_to_conv_pack;
architecture sim_to_conv_pack_arch of sim_to_conv_pack is
 begin
   process(clk)
      -- Объявления оператора process.
      variable kod_si, kod_kl, deshir_int:integer range 0 to 255;
      variable conv_kod_si_vec, conv_kl_vec:std_logic_vector (7
                                                        downto 0);
      variable deshifr_vec:std_logic_vector (7 downto 0);
      variable val_shifrkod:integer range 0 to 64567;

      -- Функция преобразования типа character в тип std_logic_vector.
          function CONV (X: character) return std_logic_vector is
                 variable RESULT : std_logic_vector (7 downto 0);
                 variable TEMP   : integer := CHARACTER'POS(X);
            begin
                 for i in RESULT'REVERSE_RANGE loop
                    case TEMP mod 2 is
                        when 0 => RESULT(i) := '0';
                        when 1 => RESULT(i) := '1';
                        when others => null;
                    end case;
                    TEMP := TEMP/2;
                 end loop;
                 return RESULT;
           end CONV;

      -- Функция преобразования типа std_logic_vector в тип integer.
         function CONV (X:std_logic_vector) return integer is
               constant XMAP   : integer  :=0;
               variable RESULT : integer :=0;
           begin
               for i in X'RANGE loop
                    RESULT:=RESULT*2;
                    case X(i) is
                        when '0' | 'L' => null;
                        when '1' | 'H' => RESULT :=RESULT+1;
                        when others    => RESULT :=RESULT+XMAP;
                    end case;
                end loop;
                return RESULT;
```

```
    end CONV;

-- Функция преобразования типа integer
-- в 8-битный тип std_logic_vector.
    function CONV8 (X :INTEGER) return std_logic_vector is
           constant L :integer :=8; -- Длина результата.
           variable RESULT :std_logic_vector (L-1 downto 0);
           variable TEMP  :integer;
      begin
           if X<0 then
               TEMP:=(2**RESULT'LENGTH)+X;
           else
               TEMP:=X;
           end if;
           for i in RESULT'REVERSE_RANGE loop
              case TEMP mod 2 is
                  when 0 => RESULT(i):='0';
                  when 1 => RESULT(i):='1';
                  when others => null;
              end case;
              TEMP:=TEMP/2;
           end loop;
           return RESULT;
    end CONV8;

-- Функция преобразования типа std_logic_vector в тип character.
    function CONV(X :std_logic_vector (7 downto 0)) return
                  character is
           constant XMAP :INTEGER :=0;
           variable TEMP :INTEGER :=0;
       begin
           for i in X'RANGE loop
              TEMP:=TEMP*2;
              case X(i) is
                  when '0' | 'L' => null;
                  when '1' | 'H' => TEMP :=TEMP+1;
                  when others    => TEMP :=TEMP+XMAP;
              end case;
           end loop;
           return CHARACTER'VAL(TEMP);
    end CONV;
```

```
        begin -- Тело оператора process.
        if clk'event and clk = '1' then
        -- Преобразование символа текста в логический вектор.
           conv_kod_si_vec := CONV(textsim);
        -- Преобразование символа ключа в логический вектор.
           conv_kl_vec := CONV(texkod);
        -- Преобразование логического вектора символа текста в код.
           kod_si := CONV (conv_kod_si_vec);
        -- Преобразование логического вектора символа ключа в код.
           kod_kl := CONV(conv_kl_vec);
        -- Шифрование путем сложения кодов текста и ключа.
           val_shifrkod := kod_si + kod_kl;
        -- Пересылка зашифрованного текста в порт shifrkod.
           shifrkod <= val_shifrkod;
        -- Дешифрование путем вычитания кода ключа из шифра.
           deshir_int := val_shifrkod - kod_kl;
        -- Преобразование дешифрованного кода
        -- в 8-битный логический вектор.
           deshifr_vec := CONV8(deshir_int);
        -- Преобразование 8-битного логического вектора
        -- в символ с отправкой результата в порт deshirtext.
           deshirtext <= CONV (deshifr_vec);
        end if;
   end process;
end sim_to_conv_pack_arch;
```

11.2.6. Подстановочный символьный криптопроцессор

Данный криптопроцессор шифрует текстовую информацию по методу подстановки (замены) символов открытого текста. В результате шифрования исходное шифруемое слово `Marino` заменено словом `Nukita`, т. е. при шифровании символ «M» замещается символом «N», символ «a» — «u» и т. д. Криптопроцессор представляет собой реализацию конечного автомата (КА), каждому состоянию которого соответствует отдельная процедура замены символа криптопроцессора.

```
-- ------------------------------------------------------
--                    Проект Zam_Simv1
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com ++++++++++++++++
-- ------------------------------------------------------
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity zam_simv1 is
   port (     clk: in  std_logic;
              rst: in  std_logic;
          simv_in: in  character;
         simv_out: out character);
end;
architecture zam_simv1_ar of zam_simv1 is
-- Определение состояний КА путем объявления
-- пользовательского перечислимого типа stm_zam_si_type.
type stm_zam_si_type is (S1, S2, S3, S4, S5, S6, S7);
signal stm_zam_si: stm_zam_si_type;
begin
 process (clk)
     begin
          if clk'event and clk = '1' then
            case stm_zam_si is
                when S1 =>
                    if simv_in = 'r' then
                         stm_zam_si <= S3;
                    elsif simv_in = 'i' then
                         stm_zam_si <= S4;
                    elsif simv_in = 'n' then
                         stm_zam_si <= S5;
                    elsif simv_in = 'o' then
                         stm_zam_si <= S6;
                    elsif simv_in <= 'M' then
                         stm_zam_si <= S7;
                    elsif simv_in = 'a' then
                         stm_zam_si <= S2;
                    end if;
                when S2 =>   -- Замена символа a символом u.
                         simv_out <= 'u';
                    if rst = '1' then
                         stm_zam_si <= S1;
                    end if;
                when S3 =>    -- Замена символа r символом k.
                         simv_out <= 'k';
```

```
                if rst = '1' then
                    stm_zam_si <= S1;
                end if;
            when S4 =>    -- Замена символа i символом i.
                    simv_out <= 'i';
                if rst ='1' then
                    stm_zam_si <= S1;
                end if;
            when S5 =>     -- Замена символа n символом t.
                    simv_out <= 't';
                if rst ='1' then
                    stm_zam_si <= S1;
                end if;
            when S6 =>     -- Замена символа o символом a.
                    simv_out <= 'a';
                if rst ='1' then
                    stm_zam_si <= S1;
                end if;
            when S7 =>     -- Замена символа M символом N.
                    simv_out <= 'N';
                if rst = '1' then
                    stm_zam_si <= S1;
                end if;
            when others =>
            null;
        end case;
      end if;
 end process;
end zam_simv1_ar;
```

11.2.7. Перестановочный битовый криптопроцессор

Проект реализует процедуры шифрования и дешифрования текстовой информации, используя метод перестановки битов. Перестановка битов осуществляется в каждом символе, содержащемся в исходном тексте, для чего тип `character` предварительно преобразуется в тип `std_logic_vector`. При этом в каждом коде символа, представленного в виде логического вектора, реализуется перестановка двух пар битов: первая пара содержит 7-й и 6-й биты, а вторая — 3-й и 2-й. Дешифрование зашифрованного логического вектора осуществляется путем обратной перестановки битов.

В этом проекте исходная информация, подлежащая шифрованию, и информация о ключах посимвольно поступает на соответствующие входные порты. Иными словами, в проекте реализован посимвольный процесс шифрования и дешифрования, при котором символ, поступивший на входной порт, сразу же шифруется и дешифруется символом ключа, а результат дешифрования тут же пересылается на выходной порт.

```
-- ------------------------------------------------------
--                      Проект Peres1_3
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
library WORK;
use WORK.Converters.all;
entity peres1_3 is
    port(               clk : in  std_logic;
                    text_in : in  character;
                     kluch1 : in  integer range 0 to 7 :=6;
                     kluch2 : in  integer range 0 to 7 :=7;
                     kluch3 : in  integer range 0 to 7 :=2;
                     kluch4 : in  integer range 0 to 7 := 3;
              text_out_std : out std_logic_vector(7 downto 0);
         shifr_text_out_std : out std_logic_vector(7 downto 0);
             desh_text_char : out character
     );
end peres1_3;
architecture peres1_3_ar of peres1_3 is
signal sig_text_in_char     : character;
signal sig_shifr_text_std   : std_logic_vector(7 downto 0);
begin
   -- Этап шифрования.
Shif_proc :
process(clk)
        variable var_text_in_std   : std_logic_vector(7 downto 0);
        variable var_text_per1_std : std_logic_vector(7 downto 0);
 begin
        if clk'event and clk = '1' then
      sig_text_in_char <= text_in;
```

```
        var_text_in_std := Conv(sig_text_in_char);
        var_text_per1_std := var_text_in_std;
      -- Шифрование путем перестановки двух пар битов.
        var_text_per1_std(kluch2) := var_text_in_std(kluch1);
        var_text_per1_std(kluch1) := var_text_in_std(kluch2);
        var_text_per1_std(kluch4) := var_text_in_std(kluch3);
        var_text_per1_std(kluch3) := var_text_in_std(kluch4);
      -- Передача зашифрованного символа на выходной порт.
        text_out_std <= var_text_in_std;
      -- Передача зашифрованного логического вектора на
       -- выходной порт с целью контроля корректности шифрования.
        shifr_text_out_std <= var_text_per1_std;
      -- Установка внутреннего сигнала, на базе которого
      -- будет выполняться дешифрование.
        sig_shifr_text_std <= var_text_per1_std;
       end if;
end process;

-- Этап дешифрования.
Deshifr_proc:
process(clk)
      variable var_deshi_text_in_std : std_logic_vector(7 downto 0);
      variable var_deshi_text_rev_per_std : std_logic_vector(7 downto 0);
 begin
       if clk'event and clk = '1' then
            var_deshi_text_rev_per_std := sig_shifr_text_std;
            var_deshi_text_in_std := sig_shifr_text_std;
         -- Обратная перестановка битов.
            var_deshi_text_rev_per_std(kluch2) :=
                                  var_deshi_text_in_std(kluch1);
            var_deshi_text_rev_per_std(kluch1) :=
                                   var_deshi_text_in_std(kluch2);
            var_deshi_text_rev_per_std(kluch4) :=
                                  var_deshi_text_in_std(kluch3);
            var_deshi_text_rev_per_std(kluch3) :=
                                   var_deshi_text_in_std(kluch4);
        -- Передача дешифрованного символа на выходной порт.
            desh_text_char <= Conv(var_deshi_text_rev_per_std);
       end if;
end process;
end peres1_3_ar
```

11.2.8. Криптопроцессор на базе сдвиговых операций

Вариант 1. Шифрование битового вектора

В этом варианте криптопроцессор построен на базе сдвиговых операций вида **sll** и **srl**. Исходный текст, а также зашифрованные и дешифрованные данные представлены типом bit_vector, поэтому дополнительные библиотеки подключать к проекту не требуется. Подробнее операции сдвига описаны в разд. 7.3.

```
-- ------------------------------------------------------
--                    Проект Bit_Shift
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------------
entity bit_shift is
    port(              clk : in  bit;
               data_bit_in : in  bit_vector (7 downto 0);
                     kluch : in  integer range 0 to 7;
             shifr_data_bit : out bit_vector (15 downto 0);
          deshifr_data_bit : out bit_vector (15 downto 0));
end bit_shift;
architecture bit_shifta of bit_shift is
begin
 process (clk)
    variable data_in_prom, data_prom_Sdv_righ: bit_vector (15 downto 0);
 begin
        data_in_prom := data_bit_in;
        data_in_prom := data_in_prom sll kluch;
        shifr_data_bit <= data_in_prom ;
        data_prom_Sdv_righ := data_in_prom  srl kluch;
        deshifr_data_bit <= data_prom_Sdv_righ;
 end process;
end bit_shifta;
```

Вариант 2. Шифрование логического вектора

В данном варианте проекта все данные представлены типом std_logic_vector. Поэтому к проекту нужно подключить пакет ieee.std_logic_1164, в котором определен этот тип, а также пакет ieee.numeric_bit, в котором определены сдвиговые операции для данных типа std_logic_vector.

```
-- ------------------------------------------------
--                   Проект StdLV_Shift
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_bit.all;
entity stdlv_shift is
    port(             clk : in  bit;
              data_bit_in : in  std_logic_vector (7 downto 0);
                    kluch : in  integer range 0 to 7;
           shifr_data_bit : out std_logic_vector (15 downto 0);
         deshifr_data_bit : out std_logic_vector (15 downto 0));
end stdlv;
architecture bit_shifta of stdlv is
begin
   process (clk)
         variable data_in_prom, data_prom_Sdv_righ :
                                    std_logic_vector (15 downto 0);
    begin
         data_in_prom :=  data_bit_in;
         data_in_prom := data_in_prom sll kluch;
         shifr_data_bit <= data_in_prom ;
         data_prom_Sdv_righ := data_in_prom srl kluch;
         deshifr_data_bit <= data_prom_Sdv_righ;
   end process;
end bit_shifta;
```

Вариант 3. Шифрование ASCII-кодов

Этот вариант криптопроцессора оперирует с ASCII-кодами и построен на использовании одного из вариантов логических методов шифрования данных типа `unsigned` с помощью сдвиговых операций **`rol`** и **`ror`** (последние для типа `unsigned` определены в пакете `ieee.numeric_std`).

```
-- ------------------------------------------------
--                   Проект Std_Shift1
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com +++++++++++++++
-- ------------------------------------------------
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity std_shift1 is
    port(              clk : in   std_logic;
               data_std_in : in   unsigned(7 downto 0);
                     kluch : in   integer range 0 to 255;
           shifr_data_std : out unsigned(7 downto 0);
         desshifr_datd_std : out unsigned(7 downto 0));
end std_shift1;
architecture std_shift1a of std_shift1 is
 begin
   process(clk)
          variable data_in_prom : unsigned(7 downto 0);
          variable data_prom_Sdv_right,data_prom_Sdv_left :
unsigned(7 downto 0);
  begin
          data_in_prom := data_std_in;
          data_prom_Sdv_left := data_in_prom rol kluch;
          shifr_data_std <= data_prom_Sdv_left;
          data_prom_Sdv_right := data_prom_Sdv_left ror kluch;
          desshifr_datd_std <= data_prom_Sdv_right;
   end process;
end std_shift1a;
```

11.2.9. Криптопроцессор с многоуровневой системой шифрования, формульный метод

В этом криптографическом процессоре информация шифруется на двух уровнях. На первом уровне применена комплексная двухступенчатая операция (нелинейное формульное выражение) для получения шифра глобального ключа *kluch_real_exp* из набора двух простых ключей (*kluch1*, *kluch2*), к которым применена операция деления. На первом уровне для получения промежуточного шифра десятичных ASCII-кодов исходных текстовых данных к ним применена нелинейная операция (операция извлечения квадратного корня).

На втором уровне для получения окончательного шифра *data_shifr_out* над промежуточным шифром десятичных ASCII-кодов исходных текстовых данных выполняется дополнительное преобразо-

вание с использованием глобального ключа. Это дополнительное преобразование реализуется посредством операции сложения. Другими словами, шифрование информации в процессоре можно представить в виде следующей формулы:

$$data_shifr_out = \sqrt{data_in} + \exp^{kluch\ 1/kluch\ 2}.$$

Дешифрование данных процессор реализует посредством формулы вида:

$$data_out = data_shifr_out^{\,2} - \exp^{-Kluch_real_\exp}.$$

Из анализа формул шифрования/дешифрования данных видно, что проект реализован посредством формульных преобразований, содержащих математические функции `sqrt()` (извлечение квадратного корня) и `exp()` (экспонента аргумента). В этой связи к проекту подключается пакет `ieee.math_real`, в котором содержатся объявления и определения всех необходимых функций.

```
-- ------------------------------------------------------
--                    Проект Shift_Sqrt
-- +++++ Copyright © 1992-2007 by Yuriy Kovtanyuk +++++
-- ++++++++++++++++++ www.ysk-books.com ++++++++++++++++
-- ------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.math_real.all;
entity shifr_sqrt is
    port(           clk: in   std_logic;
                data_in: in   integer range 0   to 255;
                 kluch1: in   integer range 0   to 255;
                 kluch2: in   real    range 0.0 to 255.0;
         data_shifr_out: out real    range 0.0 to 255.0;
               data_out: out integer range 0   to 255);
end shifr_sqrt;
architecture shifr_sqrta of shifr_sqrt is
 begin
   process(clk)
```

```
        -- Объявление вспомогательных переменных.
           variable da_in_int : integer range 0 to 255 ;
           variable da_in_real : real range 0.0 to 512.0;
           variable da_in_real_sqrt: real range 0.0 to 512.0;
           variable da_out_real: real range 0.0 to 512.0;
           variable kluch_real : real range 0.0 to 255.0;
           variable kluch_real_exp : real range 0.0 to 255.0;
           variable da_shifr_real : real range 0.0 to 255.0;
           variable da_out_real_res: real range 0.0 to 255.0;
           variable kluch_real_exp_exp : real range 0.0 to 255.0;
     begin
              da_in_int := data_in;
           -- Функция real() выполняет приведение
           -- типа integer к типу real.
              da_in_real := real(da_in_int);
              da_in_real_sqrt := sqrt(da_in_real);
              da_out_real := da_in_real_sqrt ** 2;
              kluch_real := real(kluch1)/ kluch2;
           -- Получение глобального ключа.
              kluch_real_exp := exp(kluch_real);
           -- Дешифрование глобального ключа.
              kluch_real_exp_exp := exp(- kluch_real_exp);
           -- Шифрование данных глобальным ключом.
              da_shifr_real := da_in_real_sqrt + kluch_real_exp;
              data_shifr_out <= da_shifr_real;
           -- Дешифрование шифра дешифрованным глобальным ключом.
              da_out_real_res := da_out_real - kluch_real_exp_exp;
           -- Функция integer() выполняет приведение
           -- типа real к типу integer.
              data_out <= integer(da_out_real_res);
   end process;
end shifr_sqrta;
```

11.2.10. Криптопроцессор, реализующий метод эллиптических кривых

В данном проекте для шифрования/дешифрования используется метод эллиптических кривых, при котором процедура шифрования сводится к нахождению на кривой точек, соответствующих исходному тексту, ключу и паролю. Эллиптические кривые — математический объект, который может быть определен над любым полем (конечным,

действительным, рациональным или комплексным). В криптографии обычно используются конечные поля. Эллиптическая кривая есть множество точек (x, y), удовлетворяющее следующему уравнению:

$$y^2 = x^3 + ax + b,$$

а также бесконечно удаленная точка. Для точек на кривой довольно легко вводится операция сложения, которая играет ту же роль, что и операция умножения в криптосистемах RSA (*см.* пример из подразд. 11.3.2) и Эль-Гамаля.

В реальных криптосистемах на базе эллиптических уравнений используется уравнение

$$y^2 = x^3 + ax + b \bmod p,$$

где p — простое число.

Проблема дискретного логарифма на эллиптической кривой состоит в следующем: дана точка G на эллиптической кривой порядка r (количество точек на кривой) и другая точка Y на этой же кривой, нужно найти единственную точку x, такую, что

$$Y = xG,$$

т. е. Y есть x-я степень G.

Специфический метод, выбранный для того чтобы скрыть точки, был описан Н. Коблицом (N. Koblitz). Идея заключается в том, что надо выбрать открытый ключ, используя точку на кривой. Владелец ключа имеет также секретный множитель «a». Назовем кривую E, точку P, и вычислим

$$Q = aP.$$

Открытым ключом будет последовательность — (E, P, Q). Отправитель внедряет пароль в точку S и выбирает произвольный множитель r. Пароль затем посылается владельцу в виде

$$(E, rP, S + rQ).$$

Владелец восстанавливает пароль, вычисляя

$$a(rP) = rQ.$$

Вычитая это значение из второй точки, получим *S*. (Для обеспечения безопасности кривая должна иметь много точек.)

Общая структурная схема процесса взаимодействия отправителя и получателя при обмене закрытой информацией показана на **Рис. 11.1**.

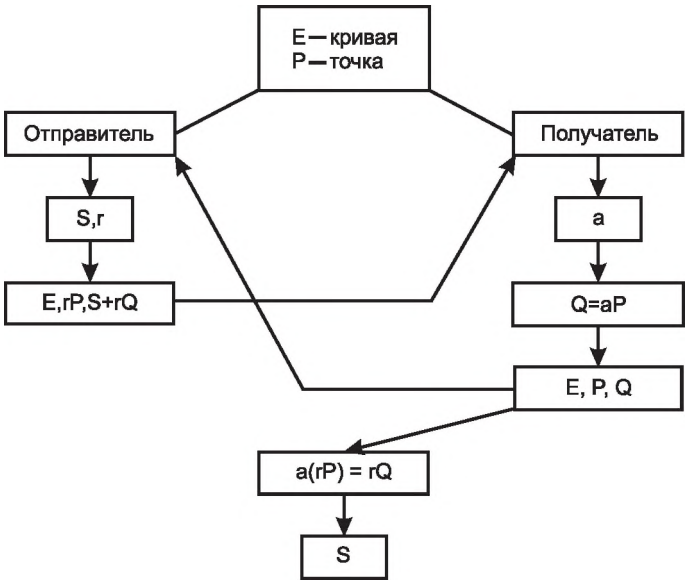

Рис. 11.1. Общая структурная схема процесса взаимодействия отправителя и получателя при обмене закрытой информацией

Более подробные блок-схемы алгоритмов процедур шифрования информации отправителем и дешифрования закрытой информации получателем показаны соответственно на **Рис. 11.2** и **Рис. 11.3**.

Процедура шифрования

Данные сначала разбиваются на блоки размером с длину ключа. Каждый блок объединяется с блоком произвольных битов для заполнения остатка поля. В рассматриваемом проекте размер блока составляет 64 бита, а размер поля — 148 бит, так что избыток составляет 84 бита. Это значение x внедряется на секретной кривой путем увеличения избыточности до тех пор, пока значение Gf_quadradic не станет удовлетворительным.

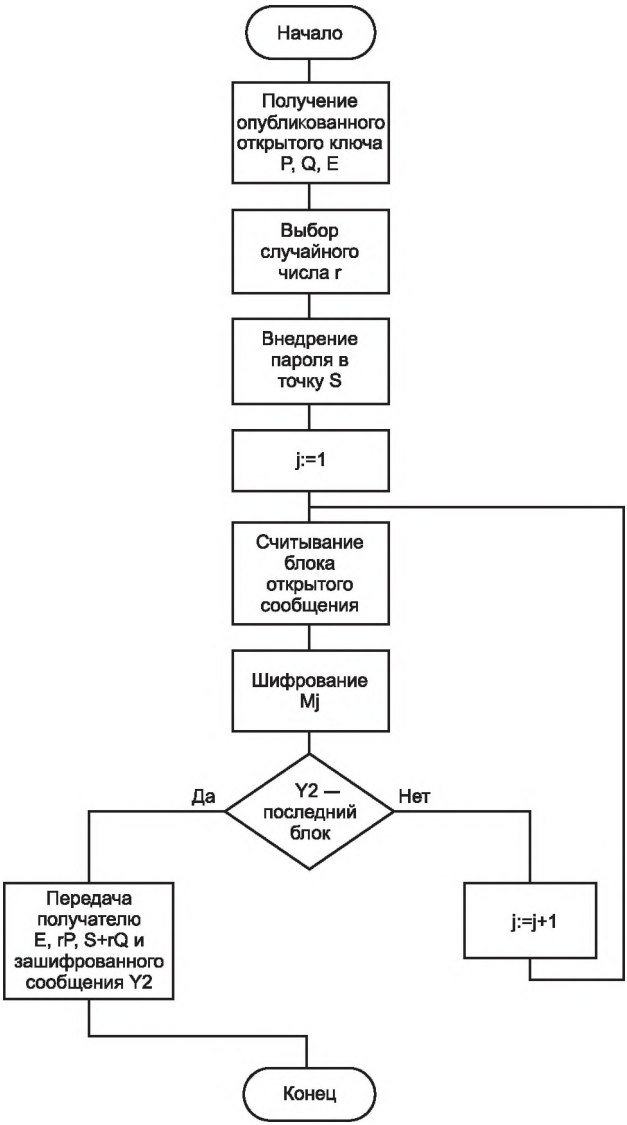

Рис. 11.2. Алгоритм шифрования отправителя

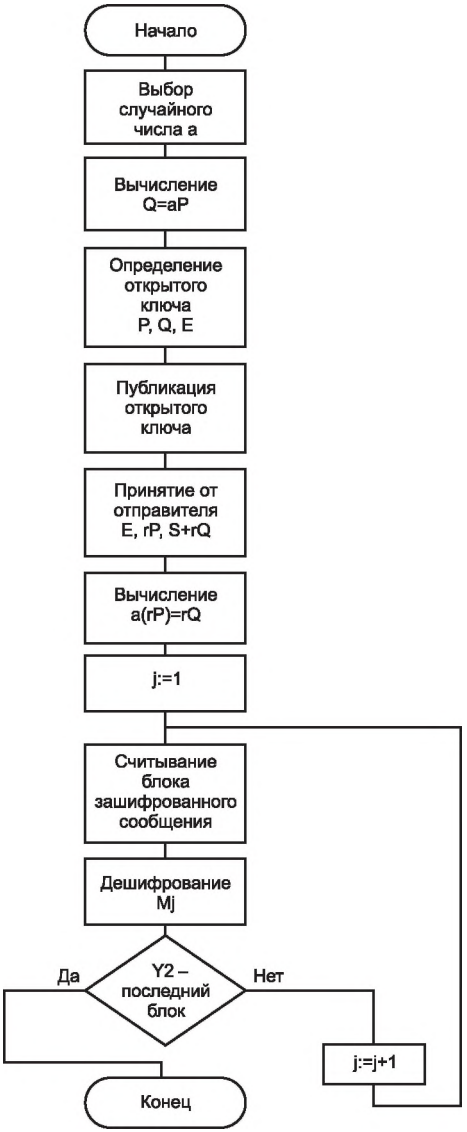

Рис. 11.3. Алгоритм дешифрования получателя

Эта точка (названная *pi* в программе) складывается с точкой шифрования и сохраняется в шифруемом блоке:

$$q_i = p_i + r_i.$$

Точка q_i «сжимается» в пределе y до 1 бита и сохраняется в шифруемом блоке.

Процедура дешифрования

Расшифровка похожа на шифрование, за исключением того, что используется один бит в пределе y для восстанавления q_i, а p_i находится с помощью вычитания:

$$r_i : p_i = q_i - r_i.$$

Только последние байты размера ключа копируются из этой точки для вывода простой текстовой области (**Рис. 11.4**).

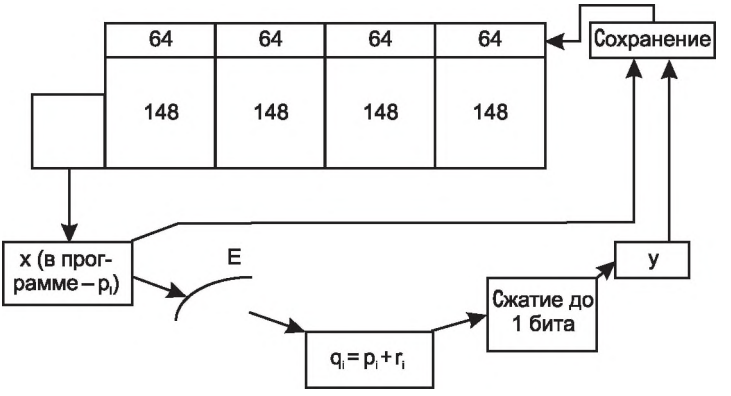

Рис. 11.4. Схема процесса дешифрования

Криптоанализ труден потому, что даже знание открытого текста и зашифрованного текста оставляют избыточность, которая больше чем ключ. В рассматриваемом примере для поля 226/128 избыточность составила 96 бит. Решение «в лоб» заключается в сравнении всех возможных избыточностей (если известен открытый текст) и прослеживании точек, которые соответствуют суммам предыдущих точек.

Листинг проекта ELIPTIC

```
library ieee;
use ieee.std_logic_1164.all;
entity eliptic is
   port (      clk : in  std_logic;
                 e : in  integer range 0 to 524287;
                 P : in  integer;
                 a : in  integer;
                 S : in  integer;
                 r : in  integer;
             kluch : in  integer;
           textkod : in  integer;
          shifrkod : out integer range 0 to 524287;
       deshifrtext : out integer)
end eliptic;
architecture eliptic of eliptic is
begin
 process(clk)
       variable kluch, kluchl, kluch2, Q, deshifr_int : integer;
       variable val_shifrkod : integer;
   begin
        if clk'event and clk=' 1 ' then
          Q :=a*P;
          kluchl := Q;
          kluch2 := S + r*Q;
          kluch := kluchl * kluch2;
          val_shifrkod := text * kluch;
          shifrkod <= val_shifrkod;
          deshifr int := val shifrkod / kluch;
          deshifrtext <= deshifr_int;
        end if;
 end process;
end eliptic;
```

Временная диаграмма работы проекта представлена на **Рис. 11.5**.

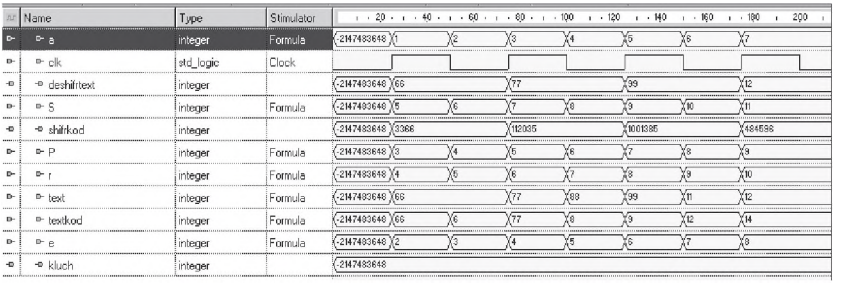

Рис. 11.5. Временная диаграмма работы проекта `Eliptic`

11.3. Проекты ЦУ специального назначения повышенного уровня сложности

11.3.1. Файловый криптопроцессор циклического сдвига

Техническое задание

Процессор должен шифровать текстовую информацию заданного объема, представленную символами ASII. Текстовая информация должна содержать фамилию, имя, отчество студента, номер группы (оставшаяся часть может быть произвольной). Текстовая информация, предназначенная для обработки в процессоре, должна храниться в созданном вами файле ОС, читаться из файла и затем перезаписываться в массив, объявленный в проекте.

Объем шифруемой информации: не менее 96 байт. Для шифрования информации применить вектор ключей, содержащий символы из случайного набора или целые числа.

Количество шифрующих ключей: 6.

Тип ключей: коды ASCII-символов.

Место хранения кодов ключей: файл операционной системы.

Коэффициент повторяемости использования ключей: 16.

Используемый метод шифрования информации: циклический сдвиг на 3 бита/ ключ.

Зашифрованный код символа должен пересылаться на выходной порт типа `std_logic_vector`.

Дешифрованный код символа должен пересылаться на выходной порт типа `character`, а также записываться в созданный файл ОС.

Описание характеристик проекта

Исходные данные для проектирования:

- объем шифруемой текстовой информации — 96 байт;
- шифруемый текст, размер которого равен 96 байт, имеет вид: `S t a k h O l g a R o m a n o v n a 5 1 9 g r . l a l i l a l i l a l i l a l i l a l i l a l i S t a k h O l g a R o m a n o v n a 5 1 9 g r . l a l i l a l i l a l i l a l i l a l i l a l i S t a k h O l g a R o m a n o v n a 5 1 9 g r . l a l i l a l i l a l i l a l i l a l i l a l i ;`
- набор ключей состоит из 6 ASCII-символов вида: `S t a k h !`
- шифрующая операция — циклический сдвиг 3 бита / ключ;
- дешифрующая операция — обратная к шифрующей.

Описание портов:

- входные порты (тип `std_logic`):
  - •`clk` — тактовый сигнал;
  - •`clk2` — тактовый сигнал, для вывода информации на выходные порты;
  - •`rd_da_fa` — сигнал разрешения чтения данных из файла;
  - •`wr_da_fa` — сигнал разрешения записи дешифрованных данных в файл;
- выходные порты:
  - •`shufr_r` — зашифрованная информация (тип `real`);
  - •`deshufr_ch` — расшифрованная информация (тип `character`).

В проекте используются три файла, которые должны быть созданы заранее с помещением путей к файлам в листинг проекта:

- `data.dat` — файл, хранящий исходную шифруемую информацию;
- `kluch.dat` — файл, содержащий информацию о шифрующих ключах;

- `desh.dat` — файл, в который помещается дешифрованная информация.

Исходная шифруемая информация и информация о шифрующих ключах, хранимая в файлах, инициализирует массив, объявленный в теле проекта.

В архитектуре проекта используются несколько переменных, которые используются в процессе преобразования типов данных в процессах их шифрования и расшифровывания.

В проекте используется стандартный библиотечный пакет `textio`, в котором определен специальный файловый тип `text`, дающий возможность получать доступ к текстовой информации, а также выполнять над этими файлами операции чтения, записи и добавления.

В проекте используется библиотечный пакет `work.converters`, содержащий описания и определения функций преобразования типов `conv();`.

Работа проекта промоделирована с помощью САПР Active HDL 5.1.

Листинг проекта FILE1_PER

```
library ieee;
use ieee.std_logic_1164.all;
use STD.TEXTIO.all;
use Work.Converters.all;
entity file1_per is
    port(
                clk : in  std_logic;
               clk2 : in  std_logic;
          rd_da_fa : in  std_logic;
          wr_da_fa  : in  std_logic;
            shufr_r : out real ;
         deshufr_ch : out character);
end file1_per;
architecture file1_per_arch of file1_per is
    signal mas1_inf,ch : string (1 to 96);
    signal sig_mas2_kl : string (1 to 6);
    type mas1_std is array (1 to 96) of std_logic_vector(0 to 7);
    type mas1_bitvec is array (1 to 96) of bit_vector (0 to 7);
    type mas1_int is array (1 to 96) of integer;
    type mas1_real is array (1 to 96) of real;
    type mas2_std is array (1 to 6)of std_logic_vector (0 to 7);
```

```
    type mas2_int is array (1 to 6)of integer;
    signal shufr_real : mas1_real;
    begin
    -- Оператор process с именем read_proc осуществляет чтение
    -- исходных данных из файла data.dat и ключа из файла -- kluch.dat,
а также инициализирует текстовый массив
    -- mas1_inf и массив ключей sig_mas2_kl.
    read_proc : process(clk)
         variable read_buf1 : string(1 to 96);
         variable wid1 : natural;
         variable read_buf2 : string(1 to 6);
         variable wid2 : natural;

         file read_cont_data : TEXT open READ_MODE is
              "C:\My_Designs\kursach2\src\data.dat";
         file read_cont_kl : TEXT open Read_Mode is
              "C:\My_Designs\kursach2\src\kluch.dat";
         begin
              if clk'event and clk ='1' and rd_da_fa ='1' then
                 FILE_OPEN(read_cont_data,
                      "C:\My_Designs\kursach2\src\data.dat",
                      READ_MODE);
                 READ(read_cont_data, read_buf1, wid1);
                 -- Инициализация текстового массива.
                 mas1_inf <= read_buf1;
                 FILE_OPEN(read_cont_kl,
                      "C:\My_Designs\kursach2\src\kluch.dat",
                      READ_MODE);
                 READ(read_cont_kl, read_buf2, wid2);
                 -- Инициализация массива ключей.
                 sig_mas2_kl <= read_buf2;
              end if;
              FILE_CLOSE(read_cont_data);
              FILE_CLOSE(read_cont_kl);
end process;

-- Данный оператор process выполняет
-- шифрование и дешифрование.
process (clk)
         variable i1,i2:positive;
         variable deshufr1_std, deshifr_data_stdvec: mas1_std;
```

```
variable deshufr1_bit_vec,deshifr_data_bit_vec : mas1_bitvec;
variable deshufr1_int:mas1_int;
variable mas1_inf_std, shufr1_std_vec :mas1_std;
variable mas1_inf_bit_vec,shufr1_bit_vec : mas1_bitvec;
variable shufr1_int : mas1_int;
variable deshufr1_real : mas1_real;
variable mas2_kl_std :mas2_std;
variable mas2_kl_int : mas2_int;
begin
     if clk'event and clk = '1' then
         for i1 in 1 to 96 loop
             mas1_inf_std(i1):=CONV(mas1_inf(i1));
             mas1_inf_bit_vec (i1):=CONV (mas1_inf_std (i1));
             shufr1_bit_vec (i1):= mas1_inf_bit_vec(i1) rol 3 ;
             shufr1_std_vec(i1) := CONV (shufr1_bit_vec(i1));
             shufr1_int(i1) := CONV (shufr1_std_vec(i1));
         end loop;
         for i2 in 1 to 6 loop
             mas2_kl_std(i2) := CONV(sig_mas2_kl(i2));
             mas2_kl_int(i2):=CONV(mas2_kl_std(i2));
         end loop;
         for i1 in 1 to 96 loop
             shufr_real(i1) <=
                real(shufr1_int(i1))/real(mas2_kl_int(i2));
             deshufr1_real(i1) :=
                real(mas2_kl_int(i2))*shufr_real(i1);
             i2:=i2+1;
             if i2>6 then i2:=1;
             end if;
         end loop;
         for i1 in 1 to 96 loop
             deshufr1_int(i1) :=integer (deshufr1_real(i1));
             deshufr1_std(i1) := CONV8 (deshufr1_int(i1));
             deshufr1_bit_vec(i1) := CONV (deshufr1_std(i1));
             deshifr_data_bit_vec(i1) :=
                deshufr1_bit_vec(i1) ror 3;
             deshifr_data_stdvec(i1) :=
                CONV(deshifr_data_bit_vec(i1));
             ch(i1)<=CONV (deshifr_data_stdvec (i1));
         end loop;
     end if;
```

```
end process;
process (clk2)
     variable i1: positive;
     variable deshifr_data_stdvec: mas1_std;
  -- variable shufr_real : mas1_real;
     begin
          if clk2'event and clk2 = '1' and i1<=96 then
              shufr_r <= shufr_real (i1);
              deshufr_ch<= ch(i1);
              i1:=i1+1;
          end if;
end process;
-- Оператор process с именем write_proc записывает
-- дешифрованные данные в файл desh.dat.
write_proc : process(clk)
variable write_buf : string(1 to 96);
file write_contejner : TEXT open write_MODE is
          "C:\My_Designs\kursach2\src\data.dat";
begin
          if clk'event and clk ='1' and wr_da_fa ='1' then
          -- Инициализация буферного массива массивом ch
              write_buf := ch;
              FILE_OPEN(write_contejner,
                   "C:\My_Designs\kursach2\src\desh.dat",
                   write_MODE);
              WRITE(wtite_contejner, write_buf);
          end if;
          FILE_CLOSE(write_contejner);
   end process;
end file1_per_arch;
```

Временные диаграммы работы проекта показаны на **Рис. 11.6**.

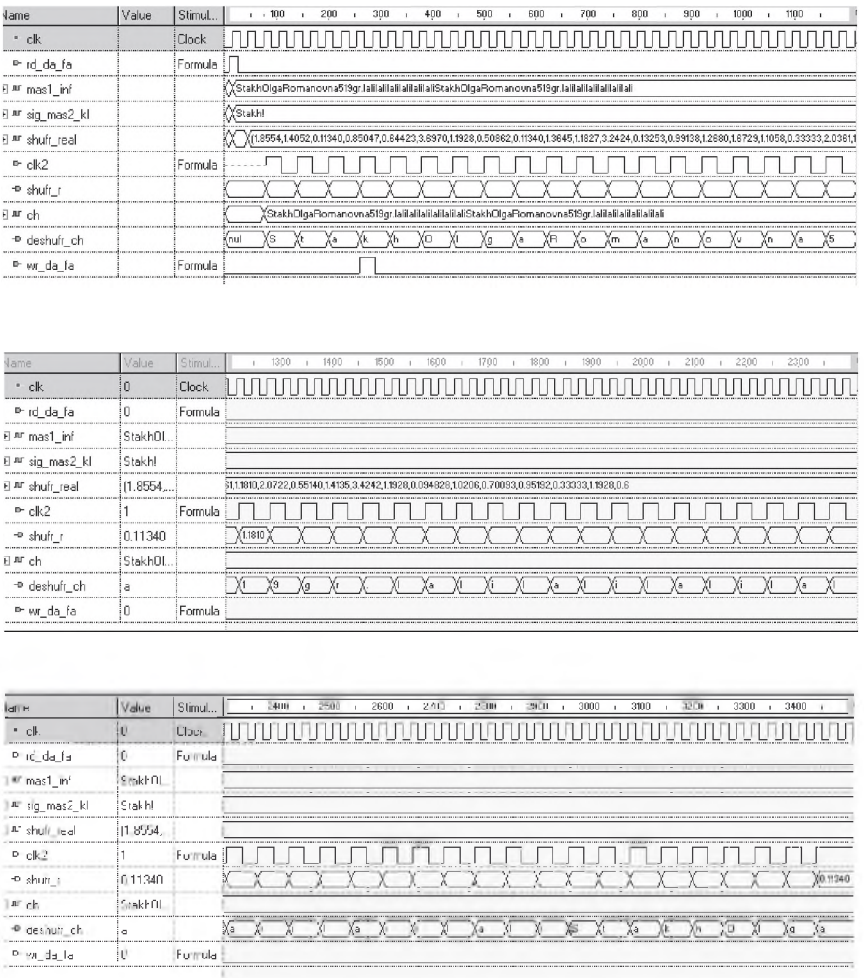

Рис. 11.6. Временные диаграммы работы проекта `File1_Per`

11.3.2. Файловый RSA-криптопроцессор

Техническое задание

Процессор должен шифровать текстовую информацию, представленную кодами ASCII.

Объем шифруемой информации: не менее 16 байт.

Текстовая информация должна быть произвольной.

Исходная текстовая информация в процессоре должна храниться в файлах ОС, читаться из файла и затем перезаписываться в массив, объявленный в проекте.

Дешифрованная текстовая информация в процессоре должна храниться в файлах ОС.

Используемый метод шифрования: RSA.

Зашифрованный код символа должен пересылаться на выходной порт типа `std_logic_vector`.

Дешифрованный код символа должен пересылаться на выходной порт типа `character`, а также должен быть записан в файл ОС.

Промоделировать работу проекта, используя САПР Active HDL 5.1

Построить временную диаграмму работы проекта и убедиться в правильности функционирования (полученная после дешифрования текстовая информация должна совпадать с исходной текстовой информацией, подлежащей шифрованию).

Описание метода шифрования и определения ключей

В данном проекте используется метод шифрования RSA, названный так по начальным буквам фамилий его разработчиков (Rivest, Shamir, Adleman).

Суть метода заключается в том, что для шифрования исходной информации используется один ключ, который *не является* секретным и часто называется *открытым ключом* (public key). В свою очередь для дешифровки секретного сообщения используется другой ключ, который является секретным и называется *закрытым ключом* (private key).

Можно выделить следующие три свойства метода шифрования RSA:

- дешифровать данные с помощью открытого ключа невозможно;
- ключ для дешифрования не может быть определен из открытого ключа, т. е. из ключа шифрования;
- ключи должны иметь очень большую величину, так как метод основан на трудности разложения очень больших целых чисел на простые сомножители.

Перед тем как приступить к изложению концепции метода, необходимо определить некоторые термины.

Чтобы использовать алгоритм RSA, нужно сначала сгенерировать открытый и закрытый ключи, выполнив следующие шаги:

- выбрать два простых числа — p и q (в идеале берутся очень большие числа);
- определить два произведения:

$$n = p \times q,$$

$$m = (p - 1)(q - 1);$$

- выбрать случайное целое число d, взаимно простое с m;
- подобрать такое целое число e, для которого является истинным следующее соотношение:

$$(e \times d) \bmod m = 1.$$

После произведенных действий получим:

- (d, n) — открытый ключ;
- (e, n) — закрытый ключ.

Теперь, чтобы зашифровать исходные данные с помощью *открытого ключа*, необходимо выполнить следующее:

- разбить шифруемый текст на блоки, каждый из которых может быть представлен в виде целого числа (data(i));
- зашифровать текст, рассматриваемый как последовательность чисел, по формуле

$$\text{code}(i) = (\text{data}(i) \text{ ^} d) \bmod n.$$

Для дешифрования зашифрованных данных с помощью закрытого ключа следует применить следующую формулу:

$$\text{decode}(i) = (\text{code}(i) \text{ ^ } e) \bmod n.$$

В результате будет получено множество чисел, которые представляют собой исходный текст.

Теперь, используя приведенные выше теоретические сведения, определим исходные данные, которые будут непосредственно использоваться в данном проекте.

Сгенерируем открытый и закрытый ключи, выполнив следующие шаги:

- согласно заданию $p = 3,\ q = 11$;
- вычисляем

$$n = p \times q = 33,$$

$$m = (p - 1)(q - 1) = 20;$$

- выбираем случайное целое число $d = 3$, взаимно простое с числом m;
- подбираем такое целое число $e = 7$, для которого является истинным следующее соотношение:

$$(e \times d) \bmod m = 1.$$

В итоге получаем:

- `(3, 33)` — открытый ключ;
- `(7, 33)` — закрытый ключ.

Таким образом, разрабатываемый криптопроцессор будет шифровать данные по формуле

$$\text{code}(i) = (\text{data}(i) \text{ ^3}) \bmod 33.$$

Соответственно для дешифрования зашифрованной информации необходимо будет применить формулу

$$\text{decode}(i) = (\text{code}(i) \text{ ^ } 7) \bmod 33.$$

На **Рис. 11.7** изображена блок-схема, которая отражает алгоритм работы криптографического процессора, реализующего шифрование/дешифрование данных методом RSA (согласно заданию на выполнение проекта).

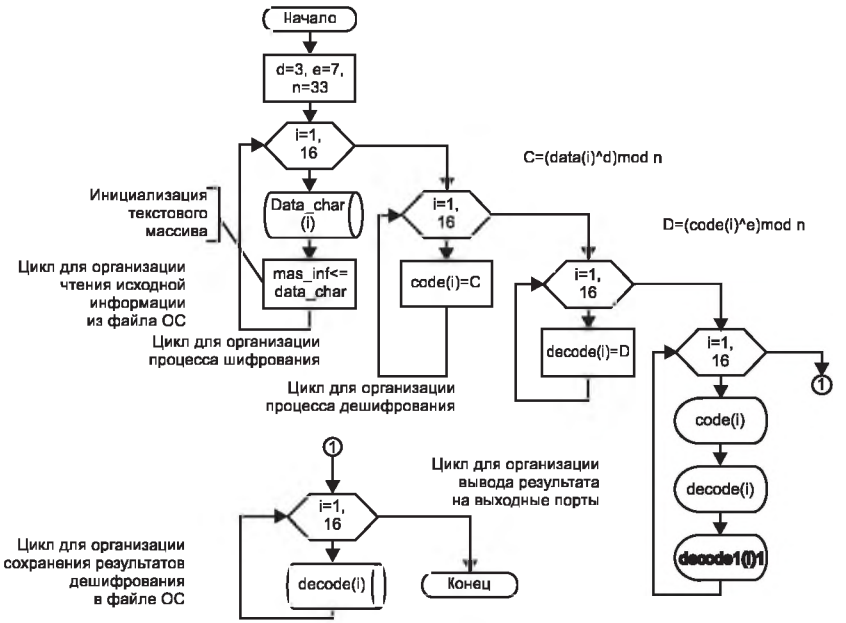

Рис. 11.7. Блок-схема шифрования по методу RSA

Описание проекта

В данном проекте используется алфавит, состоящий из 26 символов (**Табл. 11.7**). Этим символам назначаются значения кодов, приведенные в нижней строке **Табл. 11.7**.

Таблица 11.7. **Алфавит проекта Kata_RSA**

A	B	C	D	E	F	G	H	I	J	K	L	M
1	2	3	4	5	6	7	8	9	10	11	12	13

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
14	15	16	17	18	19	20	21	22	23	24	25	26

Объем информации, подлежащей шифрованию, составляет 16 байт, т. е. 16 символов. При таком варианте кодировки символов символам, входящим в шифруемый текст, будут поставлены в соответствие коды, приведенные в **Табл. 11.8**.

Таблица 11.8. **Шифруемый текст проекта Kata_RSA**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
K	A	T	Y	A	L	I	L	Y	A	O	L	U	N	I	A
11	1	20	25	1	12	9	12	25	1	15	12	21	14	9	1

В проекте используются следующие условные обозначения:

- входные порты:
  - •`clk` — тактовый сигнал;
  - •`clk_out` — тактовый сигнал для вывода информации на выходные порты;
  - •`cod` — тактовый сигнал, используемый в процессе шифрования;
  - •`decod` — тактовый сигнал, используемый в процессе дешифрования;
  - •`en_rd_f` — сигнал разрешения чтения текстовых данных из файла;
  - •`en_wr_f` — сигнал разрешения записи дешифрованной информации в файл;
- выходные порты:
  - •`code` — порт для вывода зашифрованной информации (тип `integer`);
  - •`code1` — порт для вывода зашифрованной информации (тип `std_logic_vector`);
  - •`decode` — порт для вывода дешифрованной информации (тип `character`).

Исходные текстовые данные, которые подлежат шифрованию, содержатся в файле `data.dat`. Результаты дешифрования записываются в файл `decode.dat`.

В проекте используется стандартный библиотечный пакет `textio`, в котором определен специальный файловый тип `text`, дающий возможность получать доступ к текстовой информации, а также выполнять над этими файлами операции чтения, записи и добавления.

Кроме того, в проекте используется пакет пользовательской библиотеки `my.func`, в котором содержится объявление и определение функций `conv()`, выполняющих преобразование типов данных, а также

функция возведения в степень `step()`, необходимая для возведения чисел в степень 3 и 7.

Листинг проекта KATA_RSA

```
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
library my;
use my.func.all;
entity kata_rsa is
   port(    clk : in  std_logic;
        clk_out : in  std_logic;
            cod : in  std_logic;
          decod : in  std_logic;
        en_rd_f : in  std_logic;
        en_wr_f : in  std_logic;
           code : out integer;
          code1 : out std_logic_vector (0 to 7);
         decode : out character);
end kata_rsa;
architecture arch of kata_rsa is
signal mas_inf        :  string(1 to 16);
signal sig_decod_ch   :  string(1 to 16);
signal sig_data_ch    :  string(1 to 16);
type mas_slv is array (1 to 16) of std_logic_vector(0 to 7);
type mas_int is array (1 to 16) of integer;
signal sig_cod1_slv   :  mas_slv;
signal sig_cod_int    :  mas_int;
signal sig_data_int   :  mas_int;
signal sig_decode_int :  mas_int;
begin
   read_proc: process(clk)
                    variable read_buf1 : string(1 to 16);
                    variable wid1 : natural;
                   file read_cont_data : TEXT open READ_MODE is
                        "D:\my_designs\k\src\data.dat";
              begin
                   if clk'event and clk ='1' and en_rd_f ='1' then
                      FILE_OPEN(read_cont_data,
                                "D:\my_designs\k\src\data.dat",
                                READ_MODE);
                      READ(read_cont_data, read_buf1, wid1);
```

```
                    mas_inf <= read_buf1;
                  end if;
                  FILE_CLOSE(read_cont_data);
   end process;
---------- Шифрование / дешифрование-----------
   process(clk)
          constant d: integer:=3;
          constant n: integer:=33;
          variable i: positive;
     begin
          if clk'event and clk = '1' and cod ='1' then
               sig_data_ch <= mas_inf;
            -- Шифрование с использованием
            -- открытого ключа (d,n).
               for i in 1 to 16 loop
                  sig_data_int(i) <= CONV(sig_data_ch(i));
                  sig_cod_int(i)<=(step(d,sig_data_int(i))) rem n;
                  sig_cod1_slv(i)<=CONV(sig_cod_int(i));
               end loop;
          end if;
   end process;
   process(clk)
          variable i: positive;
          constant e: integer:=7;
          constant n: integer:=33;
      begin
          if clk'event and clk = '1' and decod ='1' then
            -- Дешифрование с использованием
            -- закрытого ключа (e,n).
               for i in 1 to 16 loop
                sig_decode_int(i)<=(step(e,sig_cod_int(i))) rem n;
                sig_decod_ch(i) <= CONV(sig_decode_int(i));
               end loop;
          end if;
   end process;
---------- Вывод в порты -----------
   process (clk_out)
          variable i:positive;
     begin
          if clk_out'event and clk_out = '1' and i<=16 then
            code <= sig_cod_int(i);
            code1 <= sig_cod1_slv(i);
```

```
            decode <= sig_decod_ch(i);
            i:=i+1;
          end if;
   end process;
-- Вывод дешифрованных текстовых данных в файл ОС decode.dat.
-- При записи новых данных в файл старые данные автоматически
-- уничтожаются.
   process(clk)
          variable write_buf : string(1 to 16);
          file wtite_conteiner : TEXT open write_MODE is
               "D:\my_designs\k\src\decode.dat";
     begin
          if clk'event and clk ='1' and en_wr_f ='1' then
            write_buf := sig_decod_ch;
            FILE_OPEN(wtite_conteiner,
                      "D:\my_designs\k\src\decode.dat",
                      write_MODE);
            WRITE(wtite_conteiner, write_buf);
          end if;
          FILE_CLOSE(wtite_conteiner);
   end process;
end arch;
```

Временная диаграмма работы проекта представлена на **Рис. 11.8**.

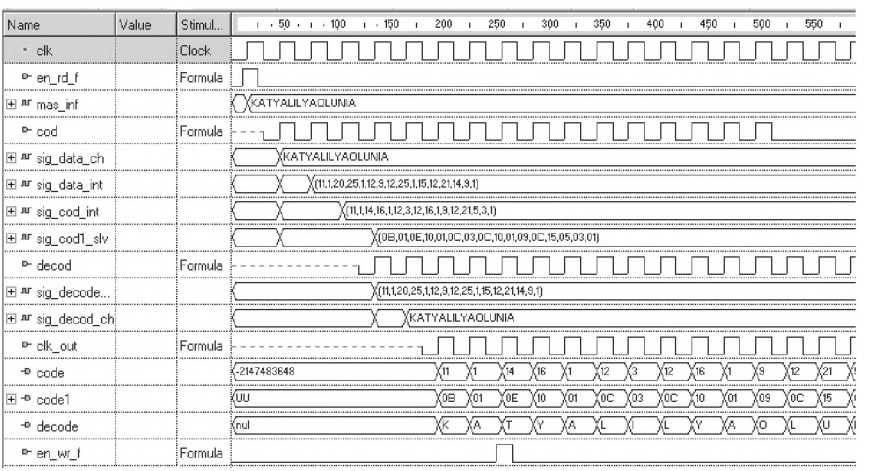

Рис. 11.8. Временная диаграмма работы проекта **Kata_RSA**

11.3.3. Файловый криптопроцессор Вижинера

Описание метода шифрования

Метод Вижинера предусматривает наличие конечной последовательности ключей

$$k = (k_0, k_1, ..., k_n),$$

которую обычно называют обобщенным *ключом пользователя*. Многократно повторяя конечную последовательность ключей, можно организовать теоретически бесконечную периодическую цепочку. Таким образом, получим *рабочий ключ*

$$k = (k_0, k_1, ..., k_n),\ k_j = k_{(j \bmod r)},\ 0 \le j < \infty.$$

Например, при $r = \infty$ и ключе пользователя 15, 8, 2, 10, 11, 4, 18 рабочий ключ будет периодической последовательностью:

15, 8, 2, 10, 11, 4, 18, 15, 8, 2, 10, 11, 4, 18, 15, 8, 2, 10, 11, 4, 18 ...

Подстановка Вижинера VIG_k определяется как

$$VIG_k : (x_0, x_1, ..., x_{n-1}) \to (y_0, y_1, ..., y_{n-1}) = (x_0+k, x_1+k, ..., x_{n-1}+k).$$

Таким образом:

- исходный текст x делится на r фрагментов:

$$x_i = (x_i, x_{i+r}, ..., x_{i+r(n-1)}),\ 0 \le i < r;$$

- i-й фрагмент исходного текста xi шифруется при помощи подстановки Цезаря Ck :

$$(x_i, x_{i+r}, ..., x_{i+r(n-1)}) \to (y_i, y_{i+r}, ..., y_{i+r(n-1)}).$$

Обобщенная система Вижинера преобразует исходный текст $(x_0, x_1, ... x_{n-1})$ в шифрованный текст $(y_0, y_1, ..., y_{n-1})$ при помощи ключа $\pi = (\pi_0, \pi_1, ..., \pi_{r-1})$ по правилу

$$VIG_k : (x_0, x_1, ..., x_{n-1}) \to (y_0, y_1, ..., y_{n-1}) = (\pi_0(x_0), \pi_1(x_1), ..., \pi_{n-1}(x_{n-1})),$$

где используется условие $\pi_i = \pi_{i \bmod r}$. Такая система, как шифр Вижинера, допускает несложную аппаратную или программную реализацию и при достаточно большой длине ключа может быть использована для реализации криптопроцессоров на базе современных ИС.

Описание проекта

Данный VHDL-проект описывает работу криптографического процессора, который предназначен для шифрования и дешифрования текстовых данных методом Вижинера (длина ключа 1). Структурная схема алгоритма, по которому в проекте реализуется процедура шифрования текстовых данных, показана на **Рис. 11.9**. Структурная схема алгоритма, по которому в проекте реализуется процедура дешифрования текстовых данных, показана на **Рис. 11.10**.

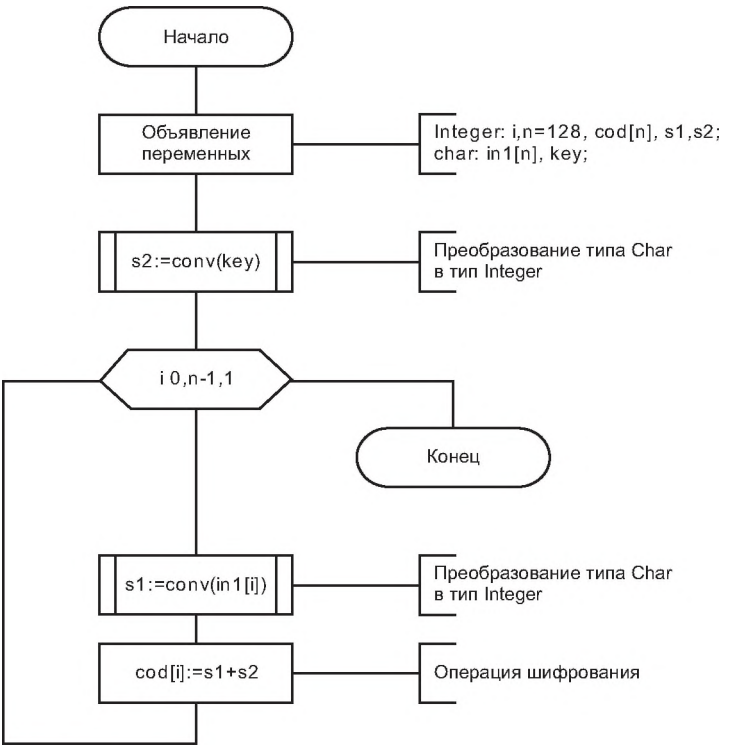

Рис. 11.9. Блок-схема шифрования по методу Вижинера

В общем алгоритме работы проекта можно выделить такие этапы, как:

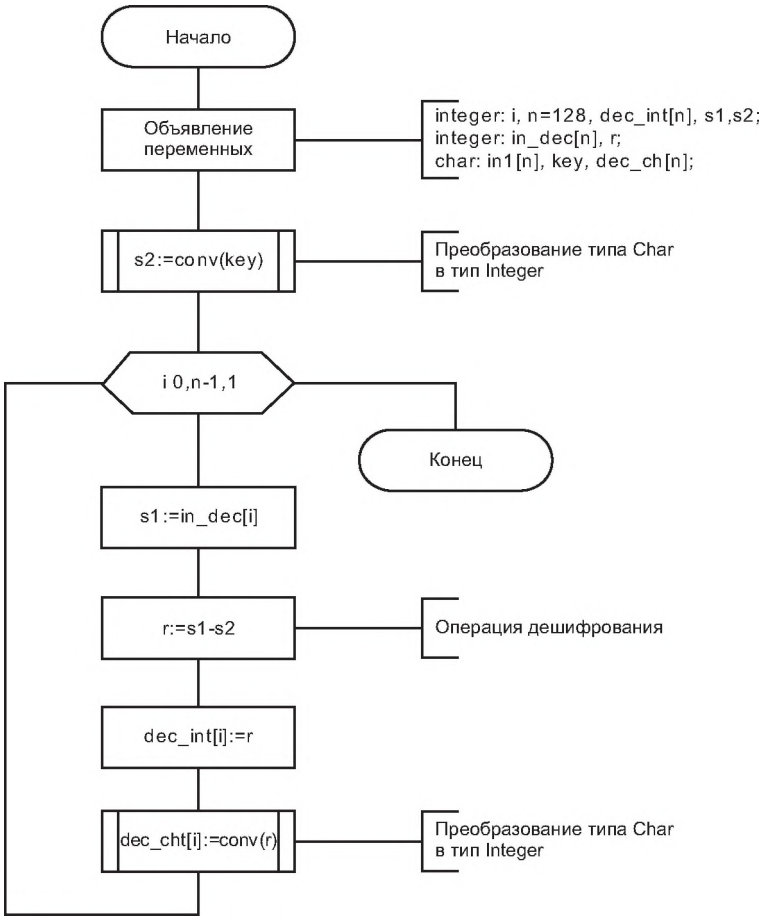

Рис. 11.10. Блок-схема дешифрования по методу Вижинера

- чтение исходного текста и текста ключа из файлов и сохранения их в массивах, объявленных в проекте;
- шифрование данных, хранимых в массивах;
- дешифрование зашифрованных данных, хранимых в массивах;
- пересылка данных (хранимых в массивах) на выходные порты;
- запись дешифрованного текста (хранимого в массиве) в файл ОС.

Шифрование и дешифрование данных осуществляется с помощью специально разработанных для этих целей пользовательских функций `coder()` и `decoder()`. Конвертирующие функции, используемые в данном проекте, хранятся в отдельном пакете `work.converters`, вследствие чего должны подключаться к проекту.

Листинг проекта Kursov2_a

```
library ieee, work;
use ieee.std_logic_1164.all, std.textio.all, work.converters.all;
entity kursov2 is
     port(    clk1:  in    std_logic;  -- Чтение данных из файла.
              clk2:  in    std_logic;  -- Чтение ключа из файла.
              clk3:  in    std_logic;  -- Шифрование.
              clk4:  in    std_logic;  -- Дешифрование.
              clk5:  in    std_logic;   -- Синхронизация вывода в порт.
     vivod_en     :  in    std_logic;  -- Запуск вывода в порт.
              cod :  out   integer;
     cod_std      :  out   std_logic_vector (7 downto 0);
              dec :  out   character;
              clk6:  in    std_logic   -- Запись дешифрованных
                                       -- данных в файл.
      );
end kursov2;
architecture kursov2_a of kursov2 is
-- Объявление и определение функции coder().
function coder (in1: character; key: character) return integer is
    variable s1, s2,q: integer;
    variable x1,x2: std_logic_vector (7 downto 0);
begin
    x1:=conv(in1);  -- Преобразование в std_logic_vector.
    s1:=conv(x1);   -- Преобразование в integer.

    x2:=conv(key);  -- Преобразовование кода ключа в std_logic_vector.
    s2:=conv(x2);
    q:=s1+s2;       -- Шифрование.
    return (q);
end coder;
-- Объявление и определение функции decode().
function decoder (y1: integer; key: character) return character is
    variable y2: integer;
    variable r: integer;
    variable dec_std: std_logic_vector (7 downto 0);
```

```
    variable key_std: std_logic_vector (7 downto 0);
    variable dec_ch: character;
begin
    key_std:=conv(key);
    y2:=conv(key_std);
    r:=y1-y2;
    dec_std:=conv8(r);
    dec_ch:=conv(dec_std);
    return (dec_ch);
end function;
--*****************************************************
signal massiv128: string (1 to 128);
signal key: character;
type rslt is array (1 to 128) of integer;
signal cod_i: rslt;
signal dec_ch1: string (1 to 128);
begin
process(clk1)
    variable read_vec: string (1 to 128);
    variable wid: natural;
    file vect_rd: TEXT open READ_MODE is "message.txt";
begin
    if clk1'event and clk1='1' then
      FILE_OPEN (vect_rd, "message.txt", READ_MODE);
      READ (vect_rd,read_vec,wid);
      massiv128 <= read_vec;
    end if;
    FILE_CLOSE(vect_rd);
end process;

process(clk2)
    variable read_vec: string (1 to 2);
    variable wid: natural;
    file vect_rd: TEXT open READ_MODE is "key.txt";
begin
    if clk2'event and clk2='1' then
      FILE_OPEN (vect_rd, "key.txt", READ_MODE);
      READ (vect_rd,read_vec,wid);
      key <= read_vec(1);
    end if;
    FILE_CLOSE(vect_rd);
end process;
```

```
process(clk3)
    variable j: integer;
begin
    if clk3'event and clk3='1' then
      for i in 1 to 128 loop
         cod_i(i)<=coder(massiv128(i),key);
      end loop;
    end if;
end process;

process(clk4)
   variable j: integer;
begin
   if clk4'event and clk4='1' then
     for i in 1 to 128 loop
        dec_ch1(i)<=decoder(cod_i(i),key);
     end loop;
   end if;
end process;

process(clk5, vivod_en)
   variable addr1: positive;
begin
   if clk5'event and clk5='1' and vivod_en='1' then
     if (addr1>=1 or addr1<=128) then
          cod<=cod_i(addr1);
          dec<=dec_ch1(addr1);
          cod_std<=conv8(cod_i(addr1));
          addr1:=addr1+1;
      else
          addr1:=1;
      end if;
    end if;
end process;

process(clk6)
   file vect_wr: TEXT open WRITE_MODE is "decode_data.txt";
begin
   if clk6'event and clk6='1' then
     FILE_OPEN (vect_wr, "decode_data.txt", WRITE_MODE);
     WRITE (vect_wr, dec_ch1);
   end if;
   FILE_CLOSE (vect_wr);
```

```
end process;
end kursov2_a;
```

Временные диаграммы работы проекта представлены на **Рис. 11.11**.

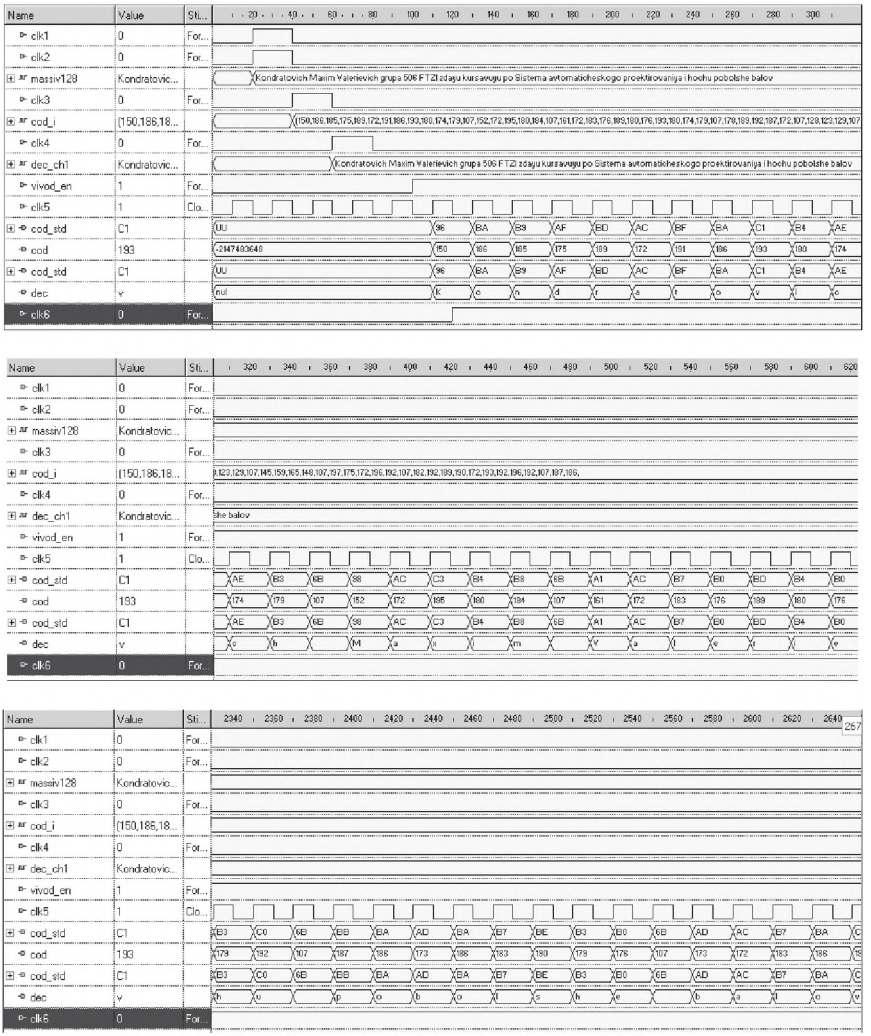

Рис. 11.11. Временные диаграммы работы проекта **Kursov2_a**

Приложение

ОСНОВЫ ЯЗЫКА VHDL В РЕФЕРАТИВНОМ ИЗЛОЖЕНИИ

П.1. Язык VHDL как универсальный язык проектирования ЦУ

Традиционно ЦУ, которые строились на логических элементах и триггерах, разрабатывались с применением булевых уравнений. Такой подход требовал при математическом моделировании записывать одно уравнение на каждый информационный выход. По мере роста масштабов цифровых систем применение булевых уравнений стало непрактичным, поскольку количество таких уравнений становится для современных ЦУ чрезвычайно велико.

Многие ЦУ содержат не только логические элементы и триггеры, но также и некоторые другие функциональные узлы цифровой схемотехники (счетчики, регистры, сумматоры и т. д.). Фактически схема, составленная из логических элементов, триггеров и функциональных узлов цифровой схемотехники, представляет собой иерархическую структуру. Представляя проект ЦУ в виде иерархической структуры, можно упростить восприятие проекта за счет инкапсуляции детализации решений на более низких уровнях иерархии.

Внедрение САПР привело к упрощению процедур проектирования относительно несложных ЦУ, в том числе и построенных на традиционных методах проектирования. Однако вскоре разработчики поняли, что схематические проекты с более чем шестью тысячами схем тяжело сопровождать и развивать даже с использованием современных САПР.

Главный недостаток традиционных методов проектирования — это ручная трансляция описания проекта в форму логических уравнений. Этот недостаток устраняется только за счет применения *языков описаний аппаратных средств* (HDL — Hardware Description Language). Используя языки программирования семейства HDL, можно разрабатывать проекты ЦУ, не заботясь о преобразовании их в форму булевых уравнений.

САПР первых поколений, основанные на языках семейства HDL, были ориентированы на анализ и проектирование ЦУ на наиболее низком уровне, непосредственно привязанном к аппаратной части разрабатываемого ЦУ. Современные языки семейства HDL позволяют создавать спецификации проектируемого ЦУ на всех уровнях — от входов и выходов базовых элементов до полнофункциональных ЦУ и сложных систем. При этом в современных САПР на каждом уровне проект может быть представлен как в структурной, так и в поведенческой форме.

Однако по мере развития САПР и средств EDA (Electronic Design Automation) стало понятно, что возникла необходимость в выработке единых форм описания проектов, поскольку на каждом уровне спецификаций можно один и тот же проект описать в разных формах. Вскоре конструкторы и разработчики ЦУ пришли к выводу, что одно из возможных решений данной проблемы состоит в создании единого языка, который бы упростил и формализовал процесс проектирования ЦУ, а также позволил бы стандартизировать документирование и передачу данных между фазами и уровнями описания проекта.

Одним из таких языков стал язык VHDL (VHSIC Hardware Description Language — *язык описания устройств на сверхбольших интегральных схемах* (СБИС)). VHDL — это универсальный язык описания аппаратных средств с поддержкой разных форм представления проектов ЦУ. Сегодня имеется множество инструментальных средств, поддерживающих синтез ЦУ по их VHDL-спецификациям.

Язык VHDL является языком параллельного программирования, т. е. является языком, способным описывать ЦУ, содержащие в своей структуре параллельно функционирующие компоненты. Иными словами, VHDL является языком, способным описывать реальную физическую работу ЦУ, в которых процессы протекают параллельно.

Язык VHDL получил широкое признание и был стандартизирован. Первая версия стандарта была издана в 1987 г. (IEEE 1076-1987). В на-

стоящее время действует стандарт VHDL, изложенный в документе IEEE 1076-2002 и являющийся промышленным стандартом, который широко используется для описания работы цифровых систем. В июне 2006 г. была опубликована версия 3.0 проекта стандарта VHDL-2006, в который вошли все дочерние стандарты, разработанные в ходе создания стандартов VHDL (IEEE 1064, 1076.2, 1076.3), а также добавлены другие усовершенствования. Стандарты VHDL, как и любые другие стандарты IEEE, пересматриваются не реже, чем один раз в пять лет.

В РФ язык VHDL закреплен стандартом ГОСТ РФ 50754-95 «Язык описания аппаратуры цифровых систем VHDL. Описание языка».

К основным достоинствам языка VHDL, которые делают его привлекательным для проектирования современных ЦУ, относятся следующие его возможности:

- Проектируемые ЦУ могут быть определены как в структурной, так и в поведенческой форме на любом уровне детализации.
- На массовом рынке ПО доступны качественные инструментальные средства проектирования и моделирования, поддерживающие язык VHDL.
- На массовом рынке ПО доступны качественные инструментальные средства синтеза VHDL- проектов.
- Язык VHDL не привязан к какой-то конкретной элементной базе.

П.2. Концептуальные положения языка VHDL

Язык VHDL поддерживает следующий принцип проектирования: каждое проектируемое ЦУ рассматривается как автономная подсистема, которая должна быть объявлена и определена. При этом любая такая подсистема может взаимодействовать со своим окружением — получать данные со своих входов, т. е. из окружающей среды, и выдавать в окружающую среду данные по своим выходам.

Часть спецификации проектируемой подсистемы, определяющей средства связи подсистемы с ее окружением, принято называть *интерфейсом*. В языке VHDL, интерфейс между системой и окружающей ее средой описывается в подразделе объявления интерфейса с помощью оператора `entity`, который является основной единицей проекта любой системы.

В языке VHDL невозможно описать ни одно проектируемое ЦУ без использования объявления интерфейса этого ЦУ с помощью соответствующего оператора **entity**.

В объявлении интерфейса определяются общие параметры ЦУ и его порты. *Общие параметры* определяют статические характеристики проектируемого ЦУ (временные задержки, тактовая частота, разрядность входных и выходных портов и т. п.). *Порты* обеспечивают каналы связи между ЦУ и его окружением. Для каждого порта должен быть определен *режим использования* (направление потока данных) и *тип* пересылаемых через него данных.

Для того чтобы проектируемое ЦУ могло выполнять возлагаемые на него функции, данные, полученные им из входных портов, должны в ЦУ подвергнутся некоторому преобразованию. Функциональное преобразование данных внутри системы реализуется в специальном подразделе проекта, который принято называть *архитектурой*. Из архитектуры проекта через выходные порты преобразованные данные передаются окружению ЦУ, определяя тем самым реакции проектируемого ЦУ на те или иные внешние факторы.

Внутри архитектуры проекта содержится описание алгоритма работы и (или) функционального устройства проектируемого ЦУ. Объявление архитектуры с помощью оператора **architecture** должно быть жестко связано объявлением интерфейса проектируемого ЦУ, т. е. — с соответствующим оператором **entity**.

Разделение проекта ЦУ на архитектуру и интерфейс позволяет использовать один и тот же интерфейс (**оператор entity**) с различными архитектурами (операторами **architecture**). Однако следует учитывать, что обратное утверждение не является истинным — каждая архитектура может быть связана только с одним интерфейсом.

В языке VHDL при создании проектов ЦУ используются две формы представления архитектуры:

- *Поведенческая (фунциональная) форма* определяет набор математических преобразований, которые должно выполнять ЦУ над входными данными, чтобы получить требуемый результат. Основой для формирования набора требуемых математических преобразований является исходное словесное описание или математическая модель ЦУ. Исходную спецификацию такого вида в языке VHDL принято называть *поведенческой*. В общем случае поведенческая форма представления определяет, *что должно делать* ЦУ

в той или иной ситуации, т. е. содержать описание *выходных* реакций ЦУ на *входные* воздействия. Внутреннее устройство ЦУ при поведенческой форме представления не имеет значения.

- *Структурная форма* определяет то, *как устроено* ЦУ, т. е. какой набор компонентов должен быть использован в ЦУ и как они должны быть связаны между собой внутри ЦУ, чтобы достичь ожидаемых результатов. Иными словами, структурная форма представления проекта регламентирует прежде всего устройство ЦУ, что не всегда позволяет однозначно сказать, какой будет реакция ЦУ на то или иное воздействие.

Язык VHDL является *расширяемым* языком. Это означает, что он позволяет использовать некоторые дополнительные конструкции, которые принято называть *пакетами* (package). Пакет — это базовая единица проектирования, предназначенная для логического объединения различных программных объектов, которые могут использоваться в разных проектах. Пакеты для большего удобства сохраняются в *библиотеках*. В языке VHDL можно использовать как пакеты, входящие в комплекты поставки САПР, так и пакеты, созданные пользователем. Это позволяет разработчику создавать и использовать собственные пакеты, подключая их к проектам по мере необходимости с помощью тех же операторов, которые используются для подключения стандартных библиотек и пакетов.

Обычно пакеты содержат два раздела: раздел объявления пакета, который определяет *интерфейс* пакета, и собственно тело пакета, в котором находятся операторы подпрограмм и значения констант, объявленных в интерфейсе пакета.

Не все пакеты содержат тело. В частности, тело пакета не нужно, если в интерфейсе пакета не объявлено ни подпрограмм, ни констант.

Для того чтобы программные объекты, объявленные в интерфейсе пакета, стали доступными в проекте ЦУ, необходимо подключить к проекту пакет и библиотеку, в которой он находится, с помощью операторов `use` и `library` соответственно.

Проект любого ЦУ может содержать разделы.

- Раздел подключения библиотек и пакетов. Является факультативным, если при описании проекта использованы исключительно встроенные типы данных, определенные в версии языка VHDL'87.

- Раздел обьявления интерфейса проектируемого ЦУ (оператор **entity**). Этот раздел является обязательным.
- Раздел архитектуры проектируемого ЦУ (оператор **architecture**, соответствующий оператору **entity**). Этот раздел также является обязательным.

П.3. Объявление интерфейса проекта

Интерфейс любого VHDL-проекта ЦУ объявляется с помощью оператора **entity**, за которым указывается идентификатор проекта. Хотя идентификатор проекта, который называют еще *именем компонента*, может быть произвольным, его выбирают таким, чтобы он соответствовал назначению проектируемого ЦУ или его компонента. При этом имя компонента в языке VHDL (как и любой другой пользовательский идентификатор) должно формироваться разработчиком по следующим правилам.

- Идентификатор должен состоять из одной строки.
- Он должен начинаться с буквы.
- Идентификатор может включать в себя: буквы, числа и символы подчеркивания. Символы подчеркивания не могут использоваться в качестве первого или последнего символа идентификатора. Кроме того, идентификатор не должен содержать два символа подчеркивания подряд.
- Внутри идентификатора пробелы не допускаются.
- Прописные и строчные символы не различаются.
- В качестве идентификаторов нельзя использовать *ключевые* и *зарезервированные* слова языка.

Комментарии в VHDL начинаются с двух дефисов (--) и заканчиваются в конце строки. Они могут быть расположены фактически где угодно, но не могут начинаться внутри идентификатора, а также ключевого или зарезервированного слова. Одна из главных причин развития языка VHDL была вызвана необходимостью наличия удобных средств для документирования проектов. Удобство и простота документирования реализуются набором комментариев, располагаемых обычно в начале каждого файла VHDL-проекта. Наличие хорошей документации — важный элемент хорошего проекта.

При объявлении интерфейса проектируемого ЦУ оператор **entity** должен заканчиваться ключевым словом **end**, за которым следует символ точки с запятой. Поскольку в VHDL имеется много конструкций и операторов, заканчивающихся именно так, рекомендуется с целью повышения удобочитаемости проекта использовать одно или оба необязательных элемента, которые могут следовать за ключевым словом **end**. Для оператора **entity** такими элементами являются ключевое слово **entity** и (или) имя компонента.

Каждый порт интерфейса проектируемого компонента определяется в операторе **entity** с помощью оператора **port**. Объявление конкретного порта содержит следующие элементы:

- необязательное ключевое слово **signal**;
- имя (т. е. идентификатор) порта;
- режим работы порта;
- тип данных, пересылаемых через порт;
- необязательное начальное значение, указываемое с помощью символов «:=»;
- символ точки с запятой, завершающий объявление порта;
- кроме того, в сложных проектах рекомендуется сопровождать объявление каждого порта комментарием, описывающим его функциональное назначение, особенности использования и т. п.

Направление потока данных через порт определяется *режимом* (mode) работы порта. Имеются три режима, которые могут использоваться для синтеза: **in** — порт получает данные извне и поэтому позволяет только *считывать* данные, генерируемые окружением компонента; **out** — порт выдает данные в окружающую среду и поэтому допускает только *записывать* в него данные, передаваемые компонентом его окружению; **inout** — порт реверсивен, т. е. допускает как чтение, так и запись данных.

В объявлении интерфейса проектируемого ЦУ, т. е. в операторе **entity**, может присутствовать конструкция **generic** (*общие_объявления*). Назначение этой конструкции — обеспечивать проектируемый компонент значениями его неизменных (для окружения компонента) параметров. Такими общими параметрами могут быть разрядность проектируемого ЦУ, его временные характеристики и т. п.

Перечень общих параметров проектируемого ЦУ располагается внутри оператора **generic**. Сам оператор **generic** располагается пе-

ред оператором `port` и содержит ключевое слово `generic`, за которым следует собственно перечень общих параметров, заключенный в круглые скобки. Каждое объявление общего параметра состоит из следующих элементов:

- идентификатор общего параметра;
- тип данных;
- символы «: =»;
- значение параметра;
- необязательный комментарий, который описывает назначение параметра.

Имена общих параметров можно использовать внутри архитектуры проекта там, где требуется использовать статические, т. е. не изменяющиеся во времени общие значения.

П.4. Программные элементы данных языка VHDL

Язык VHDL — это язык со строгим контролем типов. Это означает, что каждый используемый в проекте программный элемент данных, будь то сигнал, константа или переменная, должен иметь некоторый тип. При этом любые два идентичных по типу программных элемента данных (например, два сигнала) с разными именами рассматриваются как разные объекты.

В тех случаях, когда нужно в одном выражении использовать программные элементы данных разных типов, необходимо выполнить преобразование типа, даже если эти типы очень похожи (например, целая переменная со знаком и вещественная переменная). Преобразование типов в языке VHDL реализуется посредством специальных *функций преобразования типа* и *атрибутов*.

Каждый программный элемент данных по своей структуре может быть *простым* или *композитным*. Простой программный элемент данных не может содержать в себе более одного элемента любой сложности, поэтому такие программные элементы данных называются *скалярными*. Все значения в скалярном типе являются *перечислимыми*. Благодаря этому все операции отношений для всех простых данных скалярных типов определяются их взаимным расположением. Более того, каждое значение дискретного или физического типа имеет свое

положение на числовой оси. Все числовые типы (целое число, число с плавающей точкой и физическая величина) могут задаваться в определенном *диапазоне* (range), который состоит из набора возможных значений.

Программный элемент данных композитного типа содержит некоторое множество других объектов, которые называются элементами. Элементы могут иметь любой скалярный или составной тип. При этом не допускается использование типа `file` как элемента составного типа. В VHDL используется два вида композитных типов: *массивы* (array) и *записи* (record).

Разница между массивами и записями состоит в том, что все элементы массива должны иметь одинаковый тип, а элементы записи могут иметь различный тип. Каждый элемент массива задается именем массива и индексом. Элементы записи (называемые также *полями записи*) задаются с указанием их идентификаторов, которые следуют после имени переменной типа «запись», сопровождаемого точкой. В VHDL массивы могут быть одномерными (*векторы*) и двухмерными (*матрицы*).

Сигналы (signal) — это программные элементы данных, которые соответствуют в программном коде VHDL-проектов проводникам аппаратной части ЦУ. Сигналы в VHDL-проекте представляют собой средство связи между разными компонентами проектируемого ЦУ. Сигналы и связанные с ними VHDL-операции, такие как установка значения сигнала, задержка сигнала и т. п., используются для моделирования работы электронного оборудования, обладающего свойствами параллелизма, инерционного распространения сигнала, шины с множественными источниками управления и т. д. Для каждого сигнала ведется учет поступления новых значений по времени (хронология сигнала). Кроме того, средствами языка VHDL обеспечивается возможность установки значения каждого сигнала из разных источников. Все параметры сигнала можно получить с помощью считывания его *атрибутов* (attribute).

Проекты ЦУ, имеющие *структурную форму* представления, основаны на компонентах и сигнальных линиях, соединяющих эти компоненты. Сигнальные линии могут быть физически реализованы как:

- однопроводные линии связи. Сигналы, пересылаемые по однопроводным линиям, могут принимать в любой момент только

двоичное значение (это значение представлено в VHDL типом `bit`);

- многопроводные линии связи, которые часто называют *шинами* или *векторами*. Сигналы, пересылаемые по шинам, способны передать информацию, являющуюся комбинацией некоторого набора двоичных значений. Значения таких сигналов представляются в VHDL типом `bit_vector`, который соответствует шине с заданной разрядностью (шириной).

По месту действия в проекте сигналы могут быть классифицированы как:

- *внешние* сигналы, объявленные в интерфейсе компонента;
- *внутренние* сигналы, которые содержатся в архитектуре проекта и обеспечивают связь между компонентами субпроектов.

Каждый сигнал, используемый в VHDL-проекте, должен быть явно объявлен. Сигналы могут объявляться в:

- *пакете* — такие сигналы видимы во всех модулях проекта, использующих данный пакет;
- *архитектуре* (внутренний сигнал) — такие сигналы видимы только внутри архитектуры того проекта, в котором они объявлены;
- *блоке* — такие сигналы видимы только непосредственно внутри блока.

Объявление порта в интерфейсе проекта по существу представляет собой объявление внешнего сигнала (такое объявление называют еще неявным объявлением сигнала). Сигнал, объявленный таким способом, видим во всех архитектурах, соответствующих оператору **`entity`** проектируемого ЦУ.

При объявлении сигнала ему можно назначить начальное значение. Если в объявлении сигнала начальное значение не задано, по умолчанию используется значение, соответствующее левой границе указанного типа.

Запрещено объявлять сигналы внутри оператора **`process`** или в подпрограмме (за исключением тех случаев, когда в подпрограммах они используются как формальные параметры).

Переменные (variable) — это программные элементы данных, совместимые с сигналами. Переменные служат для описания алгоритмов работы ЦУ регистрового типа. Такие описания обычно располагаются внутри оператора **`process`**. Необходимость использования переменных вызвана тем, что описать алгоритм работы сложного ЦУ регистро-

вого типа на основании одних лишь сигналов не всегда возможно или рационально. Переменные объявляются только в разделе объявлений оператора **process** и не могут быть использованы за пределами соответствующего оператора.

Сигнал характеризуется тремя свойствами, присущими только ему: типом, значением и временем. Благодаря учету значений сигнала по времени для каждого сигнала можно получить временную диаграмму, которая называется *хронологией сигнала*. Ведение хронологии сигнала дает возможность проверить, какое значение сигнал имел раньше или когда произошло последнее изменение его значения.

Переменная характеризуется только типом и значением. Текущее значение переменной никак не связано с предыдущими ее значениями, т. е. в отличие от сигналов хронология для переменных не ведется.

В языке VHDL поддерживается широкий набор операций, каждая из которых может быть отнесена к той или иной группе приоритетности (перечислены в порядке убывания приоритета): вспомогательные операции, операции умножения, унарные операции, операции сложения, операции сдвига, операции отношений, логические операции.

Для работы с программными элементами данных стандартных типов используется пакет `standard` библиотеки `std`, который можно не подключать к проекту в явном виде. Для работы с программными объектами типа `std_ulogic`, `std_logic`, `std_ulogic_vector`, `std_logic_vector` следует к проекту подключить пакет `std_logic1164` библиотеки `ieee`. Для работы с другими типами данных следует подключить соответствующие пакеты, в которых содержатся объявления этих типов.

Операция **not** относится к вспомогательным операциям только исходя из ее приоритетности. По сути же она является логической операцией, выполняющей инверсию булевого значения.

Если используются операции **nand** или **nor**, то в соответствующих выражениях можно без скобок указывать только одну такую операцию. Если в выражении необходимо применить одну из этих операций несколько раз, следует каждую пару операндов заключать в круглые скобки.

Операция конкатенации (обозначается как `&`) объединяет два одномерных массива в массив того же самого типа большего размера (о достаточности элементов в таком массиве должен заранее позаботиться разработчик).

Имеются два механизма задержки, доступные в VHDL: инерционная задержка и транспортная задержка. Транспортная задержка определяется с помощью ключевого слова **transport** и типична для электрических проводников. Инерционная задержка определяется с помощью ключевого слова **inertial** и используется для моделирования приборов, которым присуща инерционная природа. На практике это означает, что импульсы, длина которых короче, чем указанное время переключения, не будут передаваться.

Если механизм задержки не определен, по умолчанию используется инерционная задержка. Механизмы задержек могут применяться только к сигналам.

Константа (constant) — это программный элемент данных, значение которого в процессе моделирования никогда не меняется.

Используйте константы в программном коде настолько часто, насколько это возможно, потому что они создают более наглядный и машинно-совместимый код.

П.5. Поведенческая форма проекта на основе явно заданного оператора PROCESS

Поведенческая форма представления проекта ЦУ — это описание того, *как* компонент, устройство или система преобразовывает входные данные в выходные. Иными словами, поведенческая форма — это описание набора операций, которые должны быть выполнены системой для получения нужного результата. Базовым оператором, дающим возможность реализовать поведенческую форму представления проекта, является оператор **process**. Язык VHDL поддерживает работу с операторами **process** двух классов:

- явно заданный оператор **process** (в конструкции оператора содержится ключевое слово **process**), который может иметь одну из двух форм:
  - — явно заданный оператор **process** со списком чувствительности;
  - — явно заданный оператор **process** без списка чувствительности;

- неявно заданный оператор **process** (в конструкции оператора ключевое слово **process** отсутствует). Неявно заданный оператор **process** может иметь одну из трех форм, которые обычно называют: параллельная установка значений сигналов (simple concurrent signal assignments) — ПУЗС; условная установка значений сигналов (conditional signal assignments) — УУЗС; селективная установка значений сигналов (selected signal assignments) — СУЗС.

Выполнение явно заданного оператора **process** без списка чувствительности (в силу того, что по своей природе он является циклическим) не заканчивается при выполнении последнего оператора в его теле, так как его выполнение повторяется в бесконечном цикле. Цикл выполнения этого оператора может быть приостановлен или возобновлен с помощью оператора **wait**.

Когда оператор **wait** встречается в операторе **process**, выполнение последнего приостанавливается, пока не будет выполнено условие, указанное в операторе **wait**. В языке VHDL используются следующие формы оператора **wait**:

- оператор **wait until** <*временной интервал*> — продолжение после истечения заданного временного интервала;
- оператор **wait for** <*логическое условие*> — продолжение после выполнения логического условия;
- оператор **wait on** <*список чувствительности*> — продолжение после свершения события хотя бы на одном из сигналов, перечисленных в списке чувствительности.

Возобновленный оператор **process** выполняет содержащиеся в нем операторы последовательно, пока опять не дойдет до очередного или того же оператора **wait**. Когда это происходит, процесс снова приостанавливается в ожидании выполнения условия, определенного оператором **wait**.

Оператор **wait** может появляться в любом месте оператора **process**. Обычно он используется либо в начале процесса (когда процесс нужно приостановить сразу же после запуска, *до* выполнения других операторов, содержащихся в операторе **process**), либо в конце процесса (когда нужно сначала выполнить все операторы, содержащиеся в операторе **process**, и лишь *затем* приостановить выполнение процесса).

В языке VHDL можно использовать оператор **process** с явно заданным списком чувствительности. Это позволяет определить набор сигналов или результатов, которые могут возобновлять выполнение оператора **process**. Список чувствительности указывается справа после ключевого слова **process**. Оператор **process** с явно заданным списком чувствительности эквивалентен оператору **process**, который содержит оператор **wait on** <*список чувствительности*>, указанный *последним* оператором раздела выполняемых операторов. Следует подчеркнуть, что оператор **process**, который содержит оператор **wait on** <*список чувствительности*>, указанный *первым* оператором в разделе выполняемых операторов, *не* эквивалентен оператору **process** с явно заданным списком чувствительности.

Также очень важно заметить, что оператор **process** с явно заданным списком чувствительности *не может* содержать оператор **wait** любой формы.

Новое значение сигнала, который содержится в разделе выполняемых операторов, устанавливается только после приостановки выполнения оператора **process**. При этом учитывается только то значение, которое поступило на сигнал последним за время, прошедшее от момента предыдущей приостановки процесса. Сигналы не могут быть объявлены внутри раздела объявлений оператора **process**.

П.6. Поведенческая форма проекта на основе неявно заданного оператора PROCESS

Поскольку язык VHDL является языком параллельного и ситуационного моделирования, с его помощью можно проектировать ЦУ в виде набора параллельно работающих последовательных компонентов. Каждый из этих компонентов в свою очередь может быть представлен либо в виде отдельного параллельного процесса, либо в виде отдельного параллельно-последовательного процесса, либо в виде отдельного последовательного процесса. При этом каждый отдельный компонент может функционировать параллельно с другими компонентами. Физическая имитация протекания параллельных процессов реализуется посредством специальных операторов параллельной обработки (ОПО): ОПО ПУЗС (неявно заданных операторов **process**) или ОПО

process (явно заданных операторов **process**). Эти операторы в проекте обычно содержатся в разделе ОПО архитектуры проекта.

Как и большинство составных конструкций VHDL, *архитектура проекта*, объявляемая с помощью оператора **architecture**, содержит *имя архитектуры* и идентификатор компонента, с которым связана данная архитектура. Объявление архитектуры ограничивается ключевым словом **end**. Архитектура проекта содержит два раздела: раздел объявлений и раздел выполняемых операторов. *Раздел объявлений* — это раздел архитектуры проекта, в котором прежде всего объявляется набор *внутренних сигналов*, участвующих в работе проектируемого ЦУ. Раздел выполняемых операторов — это раздел, в котором находятся ОПО, предназначенные для установки сигналов в процессе функционирования проектируемого ЦУ.

Сигналы — это единственное средство для пересылки информации между параллельно выполняющимися процессами.

В архитектуре проекта могут использоваться ОПО **process**, устанавливающие значения сигналов в явной и неявной форме. Режим выполнения определяется способностью операторов выполнить запрос на обслуживание и количеством разного вида запросов на обслуживание в текущий момент времени.

Неявно заданные операторы **process** для выполнения запроса на обслуживание должны быть чувствительны к изменению любого сигнала, который появляется в выражении, стоящем справа от символа операции установки значения сигнала (<=). Изменение значения любого сигнала в выражении активизирует выполнение операции установки.

При разработке проектов ЦУ *комбинаторного* типа установка значения сигнала должна выполняться только тогда, когда некоторое условие является истинным. Однако САПР, основанные на VHDL, не позволяют использовать для проверки истинности подобных условий конструкцию **if ... then ... else ...**, поскольку такая конструкция допустима только в явно заданном операторе **process**. Если по каким-либо причинам явно заданный оператор **process** использовать нежелательно или невозможно, следует использовать ОПО УУЗС или СУЗС. Синтаксис операторов УУЗС позволяет выполнить анализ нескольких условий и установку сигнала как при истинности всех анализируемых условий, так и при истинности одного из них. В тех случаях, когда сигналу нужно установить одно из заранее определенных зна-

чений, выбор которого зависит от значения задающего выражения, используется ОПО СУЗС. Для решения подобной задачи внутри явно заданного оператора `process` следует применять оператор `case`.

Язык VHDL позволяет обрабатывать особые ситуации, которые могут возникать при работе ЦУ. Такой особой ситуацией является, когда один и тот же сигнал одновременно устанавливается несколькими источниками. Для принятия решения о значении сигнала каждое новое устанавливаемое значение, прежде чем вступить в силу, предварительно сохраняется в ячейках специальной памяти. В языке VHDL эта процедура называется формированием значения сигнала, или управлением значением сигнала драйвером. Каждый раз, когда в проекте VHDL встречается оператор установки значения сигнала, автоматически генерируется драйвер для этого сигнала и вся информация относительно всех новых изменений значения сигнала сохраняется в драйвере сигнала. Когда выявлены все новые значения, поступившие на сигнал, они анализируются специальной подпрограммой (разрешающей функцией), которая и устанавливает окончательное значение для сигнала.

Каждый драйвер представлен временной диаграммой сигнала, состоящей из одной или нескольких последовательных транзакций. Транзакция состоит из значения сигнала и момента времени, когда сигнал принял новое значение.

П.7. Структурная форма проекта

Поведенческая форма представления определяет проектируемое ЦУ в терминах выполняемых им операций, без привязки к тому, как оно устроено. Структурная форма представления описывает именно внутреннее устройство проектируемого ЦУ: из каких компонентов оно состоит и как они связаны между собой с помощью сигналов.

Структурная форма позволяет представить проект ЦУ в виде многоуровневой иерархии компонентов, проект каждого из которых в свою очередь может быть представлен как в поведенческой, так и в структурной форме. Только компоненты самого нижнего уровня иерархии всегда должны представляться в поведенческой форме.

Компонент (component) может быть определен как отдельный ранее созданный VHDL-проект со своим интерфейсом и архитектурой,

представленными соответствующей парой операторов **entity** и **architecture** в отдельном файле проекта или пакете. Компонент может быть также определен внутри архитектуры текущего проекта, имеющего структурную форму. В любом случае, для того чтобы компонент можно было использовать в текущем проекте со структурной формой, компонент должен быть объявлен с помощью оператора **component** в разделе объявлений архитектуры текущего проекта. После этого в разделе выполняемых операторов архитектуры можно реализовывать *экземпляры* компонента.

Оператор реализации компонента **port map** — это основной оператор при использовании структурной формы представления VHDL-проектов. Этот оператор располагается в разделе выполняемых ОПО архитектуры проекта. Иными словами, оператор **port map** работает параллельно с другими ОПО проекта.

Оператор **port map** фактически реализует межкомпонентные связи экземпляра компонента с портами других компонентов, содержащихся в проекте структурной формы. Часто говорят, что оператор **port map** ассоциирует (отображает) внутренние сигналы структурного проекта с портами реализуемого экземпляра компонента.

Сигналы, используемые при реализации экземпляра компонента, могут быть как внешними портами ЦУ, так и внутренними сигналами проекта структурной формы.

В операторе **port map** реализация междукомпонентных связей может осуществляться в одной из двух следующих форм.

- Позиционная реализация порта компонента с внутренним сигналом, при котором внутренние сигналы перечислены в том же самом порядке, в котором объявлены порты в соответствующем операторе **component** компонента, экземпляр которого реализуется данным оператором **port map**.
- Именованная реализация порта компонента с внутренним сигналом, при котором внутренние сигналы связываются с портами путем указания имени порта и имени внутреннего сигнала, связанных символами =>. Обратите внимание на то, что этот символ не имеет никакого отношения к направлению информационного потока через порт.

При разработке мастер-проекта в нем допускается использовать операторы **component**, которые объявляют еще не существующие компоненты. Такие объявления игнорируются компиляторами, что

позволяет разработчику создавать проект, имеющий структурную форму, по принципу «сверху вниз», продвигаясь по мере работы над проектом от верхнего уровня иерархии к нижним.

Если проект имеет структурную форму, для его успешной компиляции нужно иметь определение компонента (пара операторов **`entity`** и **`architecture`**), объявление компонента (оператор **`component`**) и реализацию как минимум одного экземпляра компонента (оператор **`port map`**).

Оператор реализации компонентов **`port map`** относится к классу ОПО. Поэтому эти операторы выполняются по параллельному принципу и, следовательно, моделируют протекание параллельных процессов в проектах ЦУ структурной формы.