FPGA Express

Справочное руководство по VHDL

Перевод: pep

СОДЕРЖАНИЕ

Глава 1. Использование FPGA Express с VHDL

FPGA Express транслирует и оптимизирует описания VHDL во внутренний формат , эквивалентный уровню примитивных логических элементов. Этот формат затем компилируется в технологию FPGA.

Для работы с VHDL вам необходимо ознакомиться со следующими понятиями :

- Языки аппаратного описания
- VHDL
- FPGA Express
- Использование FPGA Express
- Модель процесса проектирования

Министерство обороны США разработало *VHSIC HDL* (VHDL) в 1982 году как часть программы развития сверх высокоскоростных ИС (VHSIC). VHDL описывает поведение , функции , входы и выходы цифровой схемы. VHDL , по стилю и синтаксису , аналогичен современным языкам программирования , однако включает много специфических аппаратных конструкций. FPGA Express читает и анализирует поддерживаемый синтаксис VHDL. В главе 11 перечислены все конструкции VHDL и уровни поддержки Synopsys для каждой конструкции.

Языки аппаратного описания

Языки аппаратного описания (HDL) используются для описания архитектуры и поведения дискретных электронных систем. Эти языки разработаны для построения все более и более сложных проектов. Часто проводят историческую аналогию на предмет того , как могут называться программные языки описания , от машинных кодов (транзисторы и пайка) к ассемблерным языкам (списки цепей) и далее , к языкам высокого уровня (HDL).

HDL-ориентированные системы наиболее полезны при разработке больших проектов , когда несколько разработчиков или даже несколько команд работают совместно. HDL обеспечивают структурное программирование. После того , как приняты основные архитектурные решения , а основные компоненты и связи между ними идентифицированы , работа над подпроектами может происходить совершенно независимо.

Типичные применения HDL

HDL обычно поддерживают смешанные описания , в которых структурные конструкции или списки цепей могут соединяться с алгоритмическими описаниями и описаниями поведения. При наличии таких смешанно-уровневых возможностей вы можете описывать архитектуру системы на высшем уровне абстракции ; затем проект детализируется по нарастающей для частного компонентно-уровневого выполнения. В качестве альтернативы вы можете прочитать описание проекта HDL в FPGA Express , а затем заставить компилятор автоматически синтезировать выполнение на уровне логических примитивов.

Преимущества HDL

Методология проекта , использующего HDL , имеет несколько фундаментальных преимуществ над традиционной компонентно-уровневой методологией проектирования. Среди этих преимуществ необходимо отметить следующие :

- Вы можете очень рано проверить функциональные возможности проекта и немедленно промоделировать его , записанный на HDL. Моделирование проекта на таком высшем уровне до трансляции в элементарное исполнение позволяет протестировать многие архитектурные и проектные решения.

- FPGA Express обеспечивает логический синтез и оптимизацию , так что вы можете автоматически преобразовать описание VHDL в исполнение элементарного уровня для выбранной технологии. Такая методология устраняет вышеупомянутые узкие места компонентного уровня и уменьшает время проектирования , а также количество ошибок , вносимых при ручной трансляции программы VHDL в конкретные элементы. С помощью *логической оптимизации* FPGA Express вы можете автоматически преобразовать синтезированный проект в наименьшую и наискорейшую схему. Кроме того , вы можете назначить информацию , полученную при синтезе и оптимизации схем , вновь описанию VHDL , что , возможно , приведет к более точной подстройке архитектурных решений.
- Описания HDL обеспечивают технологически независимую документацию проекта и его функциональных возможностей. Эти программы читаются и понимаются гораздо легче , чем списки цепей или схемы. Вследствие того , что начальное HDL описание проекта является технологически независимым , вы можете затем использовать его для генерации проектов с различной технологией , не затрагивая при этом оригинальную.
- VHDL , как и большинство языков высокого уровня , производит жесткую *проверку типов*. Компоненты , тип которых объявлен четырехбитовым сигналом , не могут соединяться с трех- или пятибитовым сигналом ; такое несогласование приведет к ошибке при компиляции. Если диапазон переменных определен от 1 до 15 , то ошибка возникнет при назначении им значения 0. Некорректное использование типов , как это будет показано , является основным источником ошибок при составлении описаний. Проверка типов обнаруживает такие ошибки даже перед генерацией проекта.

VHDL

VHDL является одним из многих языков аппаратного описания , широко распространенных в настоящее время. VHDL является стандартным языком HDL в соответствии с IEEE (IEEE Standard 1076 , принят в 1987) , а также Министерством Обороны США (MIL-STD-454L).

VHDL делит *объекты* (компоненты , схемы или системы) на внешние или видимые (обладающие именами и связями) и внутренние или невидимые (алгоритмы и исполнение). После того , как вы определили внешний интерфейс с объектом , другие объекты могут использовать его по мере собственного создания. Такая концепция внутреннего и внешнего взглядов является центральной при рассмотрении системных проектов VHDL. По отношению к остальным , объект определяется своим поведением и связями. Вы можете исследовать альтернативные исполнения (*архитектуры*) объекта без изменения остальной части проекта.

После того , как вы создали объект в одном из проектов , вы можете использовать его по мере необходимости в других проектах. Кроме того , вы можете создавать библиотеки объектов для применения во многих проектах или семействах проектов. Аппаратная модель VHDL показана на рис.1-1.

Объект VHDL (проект) имеет один или более входов , выходов или *портов* ввода-вывода , которые соединяются с соседними системами. Объект сам по себе состоит из совокупности соединенных между собой объектов , *процессов* и *компонентов* , которые работают совместно. Каждый объект определяется частной *архитектурой* , которая составлена из конструкций VHDL , таких как арифметические , сигнальные или компонентные реализации.

В VHDL независимые *процессы* моделируют поведение последовательных (тактируемых) схем , использующих триггера и защелки , и комбинаторных (не тактируемых) схем , использующих только логические элементы. Процессы могут определяться и называться *подпрограммами* (подпроектами). Процессы соединяются друг с другом с помощью *сигналов* (проводников). Сигнал имеет источник (формирователь) , один или более приемников и определяемый пользователем *тип* , например, «цвет» или «числа от 0 до 15».

VHDL обеспечивает широкий набор конструкций. С помощью VHDL вы можете описать дискретные электронные системы различной сложности (системы , платы , чипы , модули) с различными уровнями абстракции.

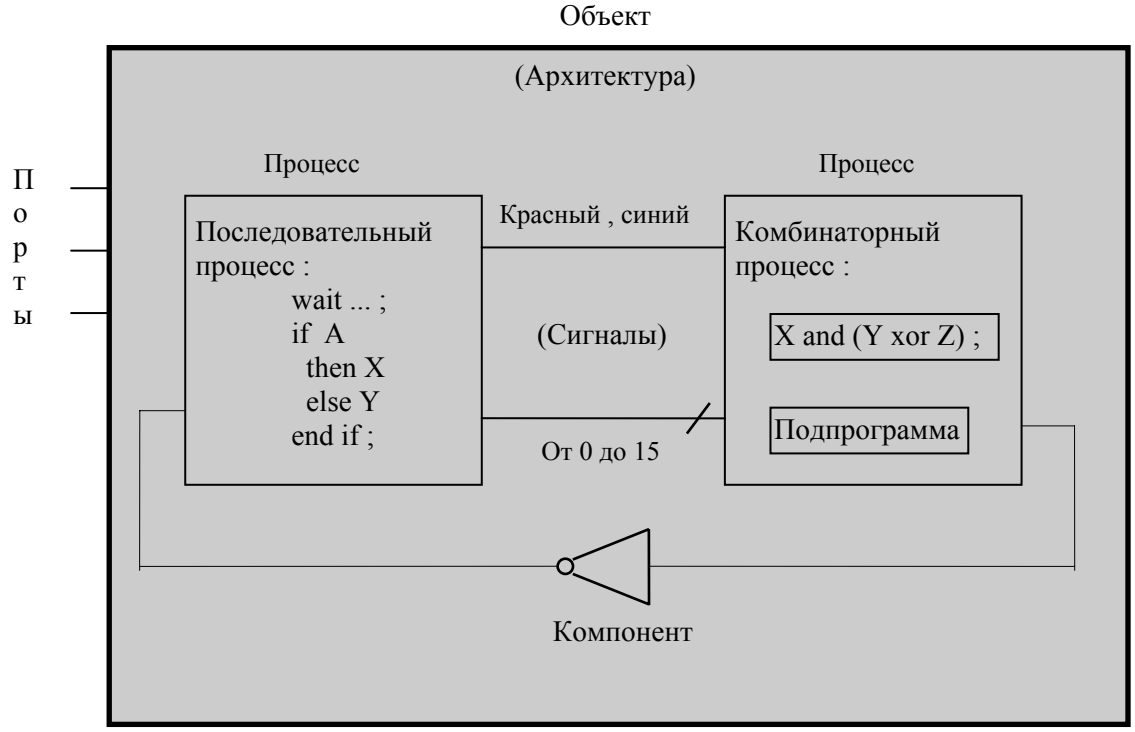


Рис.1-1 Аппаратная модель VHDL

Конструкции языка VHDL подразделяются на три категории в зависимости от уровня абстрагирования : *поведенческая , поток данных* и *структурная*. Эти категории можно описать следующим образом :

- поведенческая (behavioral)

  Функциональные или алгоритмические аспекты проекта , выраженные в виде последовательного процесса VHDL.
- поток данных (dataflow)

  Просмотр прохождения данных через весь проект , от входа к выходу. Работа в данном случае определяется в терминах набора преобразований данных , выраженных в виде параллельных утверждений.
- структурная (structural)

  Просмотр , наиболее приближенный к аппаратному обеспечению ; модель , в которой связаны все компоненты проекта. Такой просмотр выражается в виде компонентных реализаций.

Процедура проектирования FPGA Express

FPGA Express выполняет три функции :

- Транслирует VHDL во внутренний формат
- Оптимизирует представление блокового уровня с помощью различных методов оптимизации
- Размещает логическую структуру проекта в определенной технологической библиотеке FPGA.

FPGA Express синтезирует VHDL описания в соответствии с *политикой синтеза* VHDL , описанной в Главе 2 «Стили описаний». Политика синтеза Synopsys VHDL имеет три части : методология проекта , стиль проекта и языковые конструкции. Политика синтеза используется для получения высококачественных VHDL-ориентированных проектов.

Использование FPGA Express для компиляции проекта VHDL

Когда проект VHDL читается в FPGA Express , то он преобразуется во внутренний формат базы данных , так чтобы FPGA Express мог синтезировать и оптимизировать его. Когда FPGA Express оптимизирует проект , он может изменить структуру частей или всего проекта в целом. Вы управляете степенью реструктуризации. Предоставляются следующие опции :

- Полное сохранение иерархии проекта
- Разрешение перемещения целых модулей вверх или вниз по иерархии
- Разрешение комбинирования определенных модулей с другими
- Сжатие целого проекта в один модуль (*сглаживание* проекта) , если это будет полезным.

В следующем разделе описана процедура проектирования , которая использует FPGA Express вместе с симулятором VHDL.

Методология проектирования

На рис.1-2 приведена типичная процедура проектирования , которая использует FPGA Express и симулятор VHDL. Каждый шаг такой модели проектирования описан подробно.

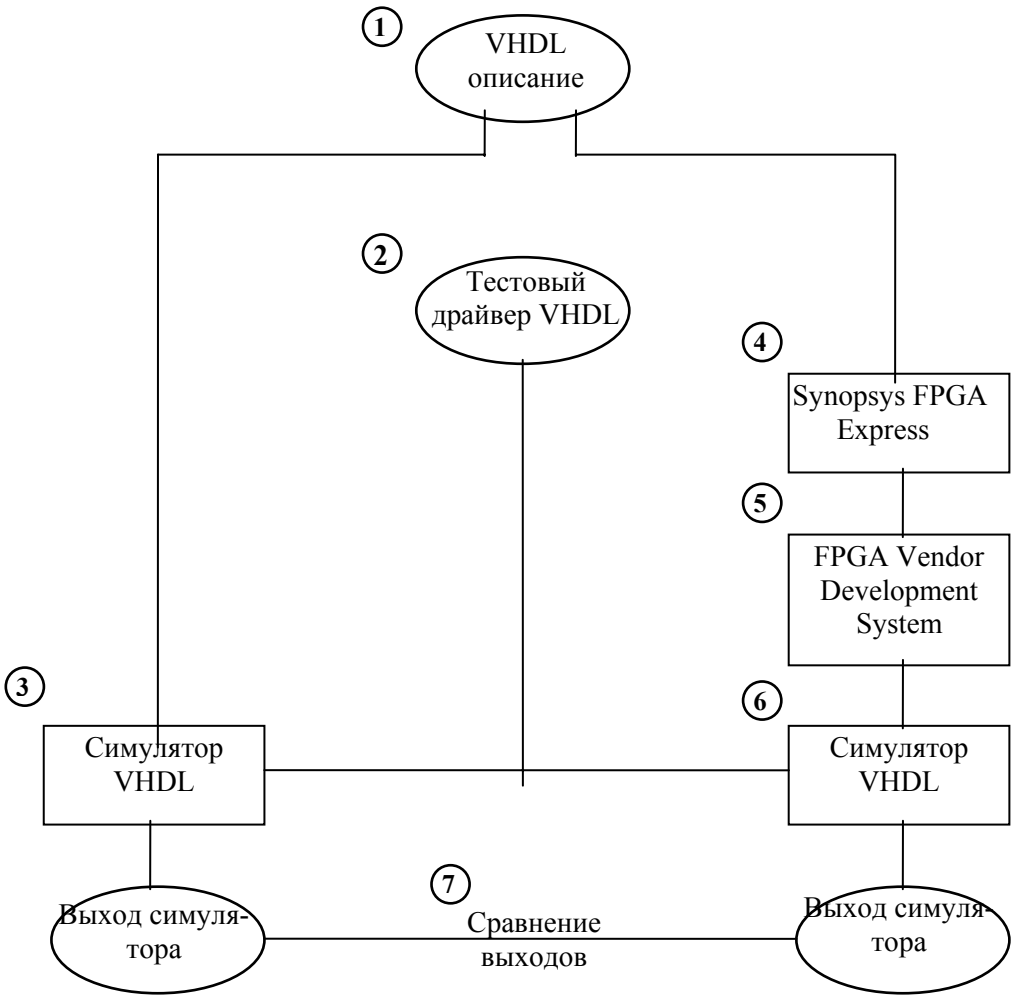


Рис.1-2. Блок-схема процесса проектирования.

1. Запись описания проекта на языке VHDL. Это описание может являться комбинацией структурных и функциональных элементов (как показано в Главе 2 «Стили описания»). Данное описание используется симуляторами FPGA Express и Synopsys VHDL.

2. Обеспечение языковых тестовых драйверов VHDL для симулятора. О правилах написания этих драйверов см. соответствующую главу в руководстве по симулятору. Драйвера обеспечивают тестовые вектора для моделирования и собирают выходные данные.
3. Моделирование проекта с помощью симулятора VHDL. Проверка правильности описания проекта.
4. Использование FPGA Express для синтеза и оптимизации описания проекта VHDL в элементарный список цепей. FPGA Express генерирует оптимизированные списки цепей , удовлетворяющие временным ограничениям выбранной архитектуры FPGA.
5. Использование вашей системы проектирования FPGA (у нас - Design Manager) для связи определенной технологической версии FPGA с симулятором VHDL. Система проектирования включает в себя модели симуляции и интерфейсы , необходимые для сквозного проектирования.
6. Моделирование определенной технологической версии проекта с помощью симулятора VHDL. Вы можете использовать оригинальные драйверы моделирования VHDL из пункта 2 , поскольку определения модулей и портов сохраняются в процессе трансляции и оптимизации.
7. Сравнение выхода моделирования на элементном уровне (пункт 6) с выходом моделирования оригинального описания VHDL (пункт 3) для проверки того , что исполнение является корректным.

Глава 2. Стили описания

Стиль вашего первоначального описания VHDL оказывает основное воздействие на характеристики результирующего проекта элементного уровня , синтезируемого с помощью FPGA Express. Организация и стиль описания VHDL определяют базовую архитектуру вашего проекта. Вследствие того , что FPGA Express автоматизирует большинство решений логического уровня , используемых в вашем проекте , вы можете сконцентрироваться на оптимизации архитектуры. Вы можете применить некоторые высокоуровневые архитектурные решения , которые невозможны без использования FPGA Express. Определенные конструкции VHDL хорошо подходят для синтеза. Для применения таких решений и использования подобных конструкций вам необходимо ознакомиться со следующими понятиями :

- Иерархия проекта
- Типы данных
- Проектные ограничения
- Выбор регистра
- Асинхронные проекты
- Языковые конструкции

Иерархия проекта

FPGA Express поддерживает иерархические границы , определяемые вами при использовании структурных конструкций VHDL. Эти границы приводят к двум основным результатам :
1. Каждый проектный объект , определяемый в описании VHDL , синтезируется отдельно и поддерживается в виде независимого проекта. При этом учитываются проектные ограничения , и каждый объект может быть оптимизирован отдельно в FPGA Express.
2. Компонентные реализации внутри описаний VHDL поддерживается в течение ввода. Имя реализации , которое вы присваиваете каждому пользовательскому объекту , сохраняется вплоть до выполнения на элементном уровне.

В Главе 3 обсуждаются объекты , а в Главе 7 - компонентные реализации.

Примечание : FPGA Express не поддерживает и не создает автоматически иерархию других неструктурных конструкций VHDL , таких как блоки , процессы , циклы , функции и процедуры. Эти элементы описания VHDL транслируются в контексте своих проектов. После чтения проекта VHDL вы можете сгруппировать вместе логику процесса , функции или процедуры внутри окна выполнения FPGA Express (Implementation Window).

Выбор иерархических границ оказывает значительное воздействие на качество синтезируемого проекта. С помощью FPGA Express вы можете оптимизировать проект , сохраняя его иерархиче-

ские границы. Тем не менее , FPGA Express только частично оптимизирует логику поперек иерархических модулей. Полная оптимизация возможна в тех частях иерархии проекта , которые сжаты в FPGA Express.

Типы данных

В VHDL вы должны назначать тип всем портам , сигналам и переменным. Тип данных объекта определяется операцией , которая может быть ему назначена. Например , оператор AND определяется для объектов типа **BIT ,** но не для объектов типа **INTEGER** . Типы данных , кроме того, важны при синтезе вашего проекта. Тип данных объекта определяет его размер (ширину в битах) и битовую организацию. Правильный выбор типов данных значительно улучшает качество проекта и помогает минимизировать количество ошибок.

См. Главу 4 , в которой обсуждаются типы данных VHDL.

Проектные ограничения

Вы можете описать качественные ограничения модуля проекта внутри FPGA Express Implementation Window. См. *Руководство пользователя FPGA Express* для более подробной информации.

Выбор регистра

Размещение регистров и тактирующих схем являются очень важными архитектурными решениями. Существует два способа определения регистров в вашем описании VHDL. У каждого из этих методов существуют определенные преимущества :

- Вы можете реализовать регистры непосредственно в описании VHDL , выбрав любой нужный элемент из библиотеки FPGA. Тактирующие схемы могут быть произвольной сложности. Вы можете выбрать архитектуру , базирующуюся на триггерах или защелках. Основными недостатками такого подхода являются :
    - Описание VHDL теперь привязано к определенной технологии , поскольку вы выбрали структурные элементы из технологической библиотеки. Однако , вы можете изолировать этот раздел вашего проекта в виде отдельного объекта , который затем будет связан с остальной частью проекта.
    - Программа пишется более сложно.
- Вы можете использовать структуры **if** и **wait** , чтобы вывести триггера и защелки из вашего описания. Преимущества такого подхода прямо противоположны недостаткам предыдущего. При использовании логического описания регистра программа VHDL становится технологически независимой и более легкой для написания. Этот метод позволяет FPGA Express выбирать тип описываемого компонента в зависимости от ограничений. Таким образом , если необходим специфический компонент , должна использоваться его реализация. Тем не менее , некоторые типы регистров и защелок не могут быть описаны логически.

  См. Главу 8 , в которой обсуждается описание регистров.

Асинхронные проекты

Вы можете использовать FPGA Express для конструирования асинхронных проектов с кратными и управляемыми синхроимпульсами. Однако , хотя эти проекты являются логически (статистически) корректными , они могут неправильно моделироваться или работать всвязи с условиями состязания.

Языковые конструкции

Другим компонентом политики синтеза VHDL является набор конструкций , которые описывают ваш проект , определяют его архитектуру и дают последовательно хорошие результаты. В оставшейся части настоящего руководства описываются именно эти конструкции и их использование.

Понятия , упомянутые ранее в данной главе , описаны в руководстве следующим образом :

Иерархия проекта

Глава 3 описывает применение и важность иерархии в проектах VHDL.

Глава 7 объясняет , как реализовать (применить) существующие компоненты.

Типы данных

Глава 4 описывает типы данных и их использование.

Выбор регистра

Вы можете реализовать регистры с помощью компонентных конструкций , описанных в Главах 3 и 7. Главы 6 и 8 описывают реализацию регистров через операторы VHDL **if** и **wait**.

Глава 3. Описание проектов

Для описания проектов в VHDL вам необходимо ознакомиться со следующими понятиями :

- Объекты VHDL
- Конструкции VHDL
- Определение проектов
- Структурные проекты

Объекты VHDL

Проекты , которые описываются в VHDL , состоят из объектов. *Объект* представляет собой один уровень иерархии проекта и может содержать полный проект , существующие аппаратные компоненты или VHDL-определенный объект.

Каждый проект состоит из двух частей : спецификации объектов и архитектуры. Спецификация объекта является его внешним интерфейсом. Архитектура объекта является его внутренним исполнением. У проекта есть только одна объектная спецификация (интерфейс) , однако он может иметь несколько архитектур (исполнений). Когда объект компилируется в аппаратный проект , то конфигурация определяет используемую архитектуру. Объектная спецификация и архитектура могут содержаться в различных исходных файлах VHDL или в одном файле. В примере 3-1 показана объектная спецификация простого логического элемента (2-входовой И-НЕ).

Пример 3-1. Объектная спецификация VHDL

```
entity NAND2 is
        port(A, B: in BIT;          -- Два входа , A и B
        Z: out BIT);                -- Один выход , Z = (A and B)'
end NAND2;
```

Примечание : В описаниях VHDL комментарии предваряются двумя дефисами (--). Все символы после этих дефисов до конца строки игнорируются FPGA Express. Единственными исключениями из этого правила являются комментарии , которые начинаются с -- pragma или -- synopsys ; эти комментарии являются директивами FPGA Express.

Ключевое слово **entity** объявляет объектную спецификацию , этот объект может использоваться другими объектами проекта. Внутренняя архитектура объекта определяет его функцию.

В примерах 3-2 , 3-3 и 3-4 показаны три различных архитектуры для объекта **NAND2** . Три примера определяют эквивалентное исполнение **NAND2**. После оптимизации и синтеза каждое исполнение приводит к одинаковой схеме , вероятно к 2-входовому элементу И-НЕ выбранной технологии. Стиль описания архитектуры , используемый вами для данного объекта , зависит только от собственного предпочтения. В примере 3-2 показано , как объект **NAND2** может быть выполнен с помощью двух компонентов из технологической библиотеки. Входы объекта **A** и **B** соединяются с элементом И **U0** , что дает промежуточный сигнал **I**. Сигнал **I** затем соединяется с инвертором **U1** , обеспечивая выход объекта **Z**.

Пример 3-2. Структурная архитектура для объекта NAND2

```
architecture STRUCTURAL of NAND2 is
        signal I: BIT;

        component AND_2       -- Из технологии
library
                port(I1, I2: in BIT;
                O1: out BIT);
        end component;

        component INVERT      -- Из технологии
library
                port(I1: in BIT;
                O1: out BIT);
        end component;
begin
        U0: AND_2 port map (I1 => A, I2 => B, O1 => I);
        U1: INVERT port map (I1 => I, O1 => Z);
end STRUCTURAL;
```

В примере 3-3 показано , как вы можете определить объект **NAND2** с помощью его логической функции.

Пример 3-3. Архитектура потока данных для объекта NAND2

```
architecture DATAFLOW of NAND2 is
begin
        Z <= A nand B;
end DATAFLOW;
```

Пример 3-4. RTL архитектура для объекта NAND2

```
architecture RTL of NAND2 is
begin
        process(A, B)
        begin
                if (A = '1') and (B = '1') then
                        Z <= '0';
                else
                        Z <= '1';
                end if;
        end process;
end RTL;
```

Конструкции VHDL

Высокоуровневые конструкции VHDL совместно работают для описания проекта. Описание состоит из :

Объектов
: Интерфейсы с другими проектами.

Архитектур
: Исполнения объектов проекта. Архитектуры могут определять связи с другими объектами через реализацию.

Конфигураций
: Привязки объектов к архитектурам.

Процессов
: Наборы последовательно выполняемых команд. Процессы объявляются внутри архитектур.

Подпрограмм
: Алгоритмы , которые могут быть использованы более чем в одной архитектуре.

Блоков объявлений
: Наборы объявлений , используемые одним или более проектами.

Объекты

Проект VHDL состоит из одного или более объектов. Объекты имеют определенные входы и выходы и выполняют определенную функцию. Каждый проект состоит из двух частей : объектной спецификации и архитектуры. Объектная спецификация определяет входы и выходы проекта , а архитектура определяет его функцию.

Вы можете описать проект VHDL в виде одного или нескольких файлов. Каждый файл состоит из объектов , архитектур или блоков. Блоки определяют глобальную информацию , которая может использоваться различными объектами.

На рис.3-1 показана блок-схема иерархической организации проекта VHDL по файлам.

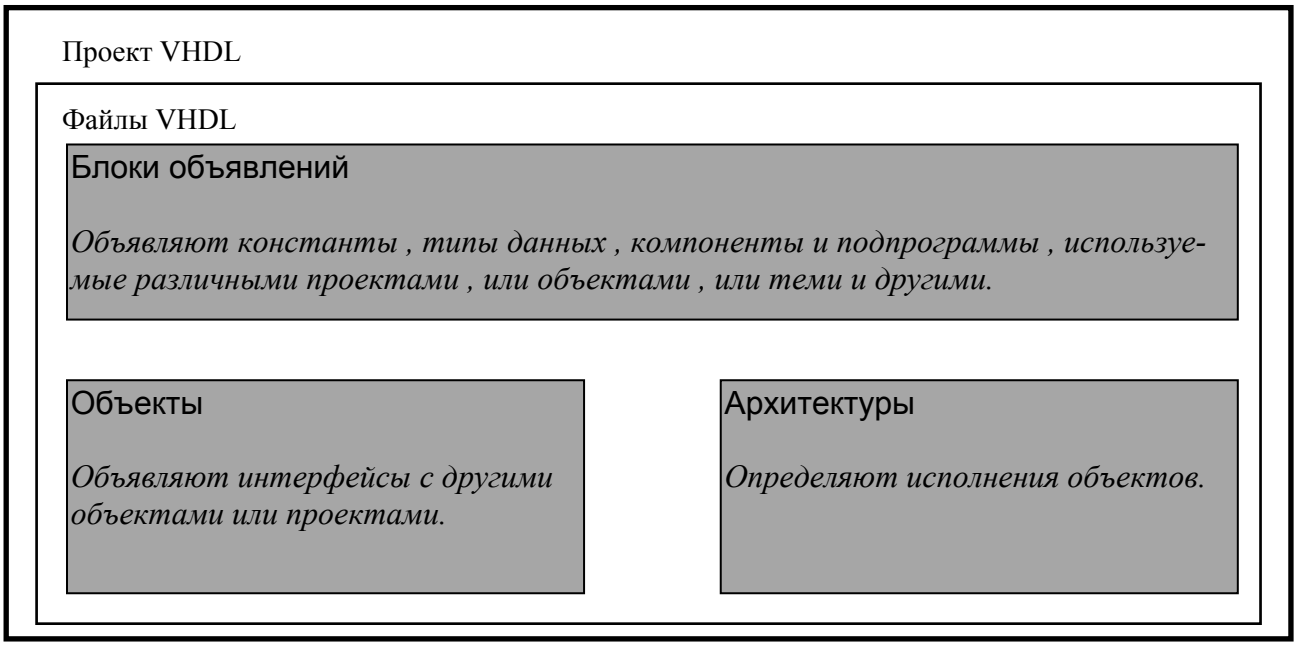


Рис.3-1. Организация проекта

Архитектуры

Архитектура определяет функцию объекта. На рис.3-2 показана организация архитектуры. Не все архитектуры содержат каждую из приведенных конструкций.

Рис.3-2. Организация архитектуры.

Архитектура состоит из раздела объявлений , где вы объявляете сигналы , типы , константы , компоненты и подпрограммы , вслед за которым следует набор параллельных операторов.

Сигналы соединяют отдельные куски архитектуры (параллельные операторы) друг с другом , а также с внешним миром - через порты интерфейса. Вы объявляете тип каждого сигнала , который означает тип данных , переносимых этим сигналом. Типы , константы , компоненты и подпрограммы, объявляемые в архитектуре , являются для нее локальными. Для использования этих объявлений более , чем в одном объекте или архитектуре , размещайте их в блоках объявлений , которые будут описаны позже в этой главе.

Каждый параллельный оператор в архитектуре определяет вычислительную единицу , которая читает сигналы , осуществляет вычисления на базе сигнальных значений и назначает вычисленные значения сигналам. Параллельные операторы вычисляют все значения одновременно. Хотя порядок параллельных операторов не влияет на порядок выполнения , операторы часто координируют свою работу посредством связи друг с другом через сигналы.

Существует пять типов параллельных операторов - блоки , сигнальные назначения , вызовы процедур , компонентные реализации и процессы. Их можно описать следующим образом :

блоки

Группируют вместе набор параллельных операторов.

сигнальные назначения

Назначают вычисленные значения сигналам или портам интерфейса.

вызовы процедур

Вызывают алгоритмы , которые вычисляют и присваивают значения сигналам.

компонентные реализации

Создают экземпляр объекта , связывая его интерфейсные порты с сигналами или портами интерфейса уже определенного объекта. См. раздел «Структурное проектирование» ниже в этой главе.

процессы

Определяют последовательные алгоритмы , которые читают значения сигналов и вычисляют новые значения для присваивания другим сигналам. Процессы обсуждаются в следующем разделе.

Параллельные операторы описаны в Главе 7.

Конфигурации

Конфигурация определяет одну комбинацию объекта и соответствующей ему архитектуры.

Примечание : FPGA Express поддерживает только конфигурации , которые соответствуют одному объекту высшего уровня с архитектурой.

Процессы

Процессы содержат *последовательные операторы* , которые определяют алгоритмы. В противоположность параллельным операторам последовательные выполняются по порядку. Этот порядок позволяет вам выполнять пошаговые вычисления. Процессы читают и записывают значения сигналов и портов интерфейса для связи с остальной архитектурой и приложениями системы.

На рис.3-3 показана организация конструкций процесса. Процессы могут содержать не все перечисленные конструкции.

Процессы являются уникальными в том , что они ведут себя по отношению к остальной части проекта так же , как параллельные операторы , однако внутри они организованы последовательно. Кроме того , только процессы могут определять переменные для хранения промежуточных значений при последовательных вычислениях.

Поскольку операторы в процессах выполняются последовательно , существует несколько конструкций для управления порядком выполнения , например , операторы **if** и **loop**. Последовательные операторы описаны в Главе 6.

Подпрограммы

Подпрограммы , так же как процессы , используют последовательные операторы для определения алгоритмов , которые вычисляют значения. В противоположность процессам , однако , они не могут непосредственно читать или записывать сигналы остальной части архитектуры. Все связи осуществляются через интерфейс подпрограммы ; каждый вызов подпрограммы имеет свой собственный набор интерфейсных сигналов.

Существует два типа подпрограмм - функции и процедуры. Функция непосредственно возвращает одиночное значение. Процедура возвращает ноль или более значений через свой интерфейс. Подпрограммы являются полезными , если вам нужно выполнить повторяющиеся вычисления , особенно в различных частях архитектуры. Подпрограммы описаны в Главе 6.

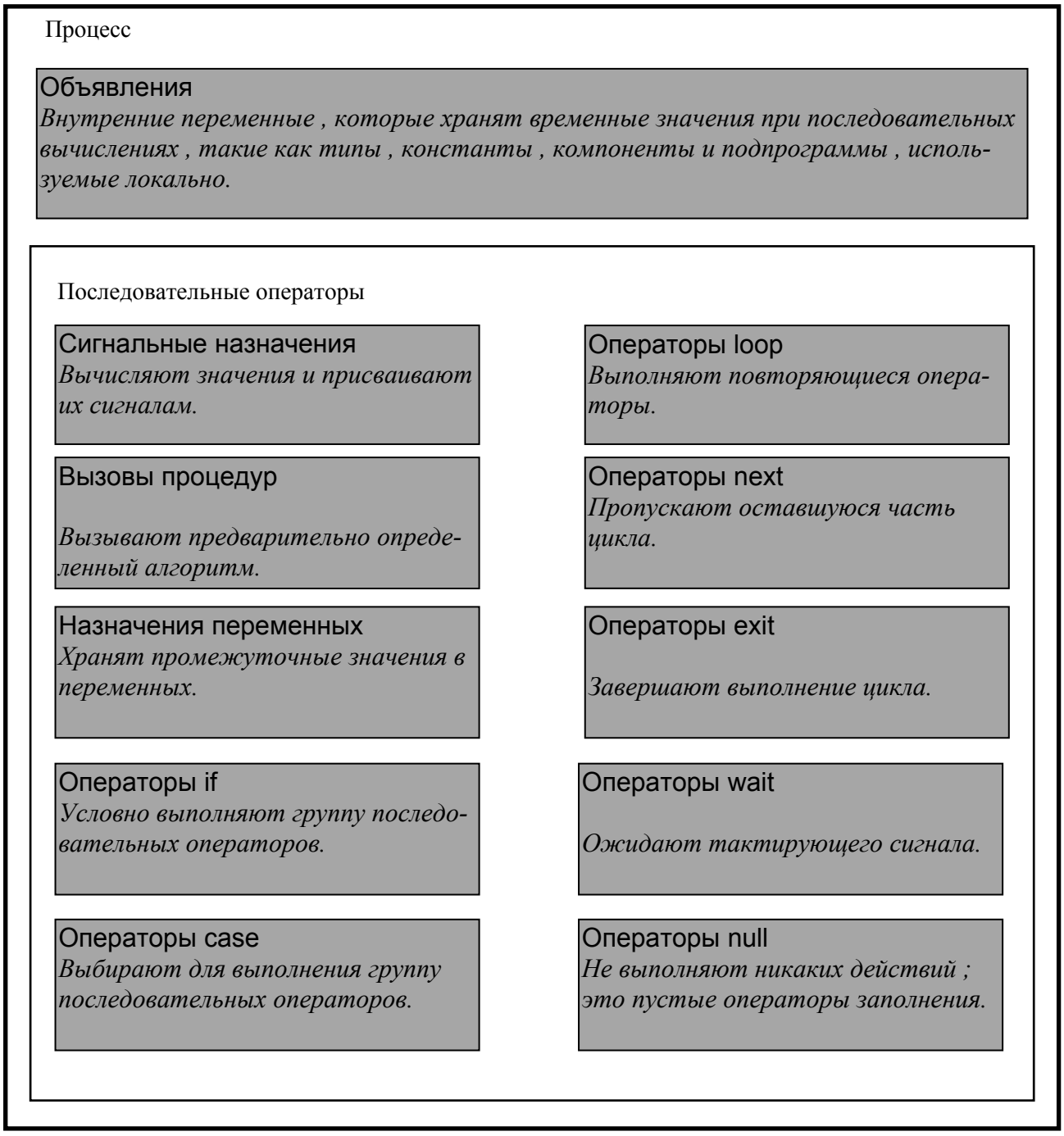


Рис.3-3. Организация процесса.

Блоки объявлений

Вы можете объединить константы , типы данных , объявления компонентов и подпрограмм в блоке объявлений VHDL , который затем будет использоваться более , чем в одном проекте или объекте. На рис.3-4 показана типичная организация блока объявлений. Блок объявлений должен содержать как минимум одну из конструкций , перечисленных на рис.3-4 :

- Константы в блоках часто объявляют параметры системного уровня , такие как ширина информационного канала.
- Объявления типов данных VHDL часто включаются в блок объявлений для объявления типов данных , используемых во всем проекте. Все объекты проекта должны иметь общие типы интерфейсов , например , общие типы адресной шины.

- Компонентные объявления определяют интерфейсы с объектами , которые могут быть реализованы в проекте.
- Подпрограммы определяют алгоритмы , которые могут быть вызваны из любого места проекта.

Блоки объявлений обычно являются достаточно общими , так что вы можете использовать их в различных проектах. Например , блок **std_logic_1164** определяет типы данных **std_logic** и **std_logic_vector** .

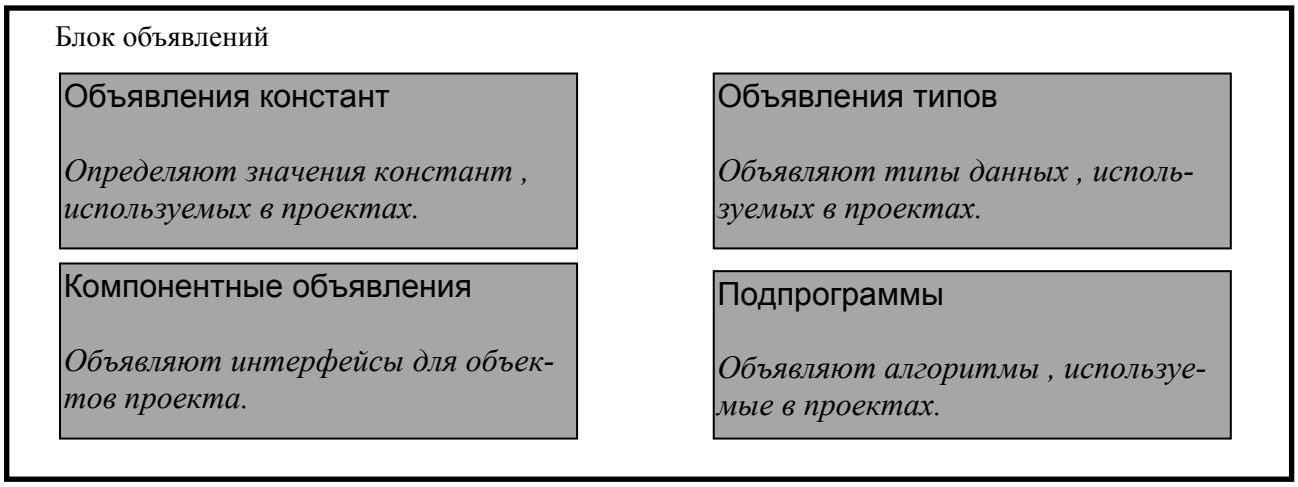


Рис.3-4. Типичная организация блока объявлений.

Использование блока объявлений

Оператор **use** позволяет объекту использовать объявления из блока. Поддерживаемый синтаксис этого оператора следующий :

use *LIBRARY_NAME.PACKAGE_NAME*.ALL; где

LIBRARY_NAME - имя библиотеки VHDL , а ***PACKAGE_NAME*** - имя включаемого блока объявлений. Оператор **use** обычно является первым в файле блока объявлений или спецификации объекта. Synopsys не поддерживает разные блоки объявлений с одинаковыми именами , если они существуют в различных библиотеках. Никакие два блока не могут иметь одинаковое имя.

Структура блока объявлений

Блок объявлений состоит из двух частей - *объявления* и *тела* :

объявление блока

Хранит *общую* информацию , включая объявления констант , типов и подпрограмм.

тело блока

Хранит *частную* информацию , включая локальные типы и исполнения подпрограмм (тела).

Примечание : Если объявление блока содержит объявления подпрограмм , то соответствующее тело блока должно определять тела подпрограмм.

Объявления блока

Объявления блока содержат информацию , необходимую одному или более объектам проекта. Эта информация включает объявления типов данных , объявления сигналов , объявления подпрограмм и объявления компонентов.

Примечание : Сигналы , объявленные в блоках , не могут совместно использоваться разными объектами. Если два объекта используют сигнал из данного блока , то каждый из них должен создавать свою собственную копию этого сигнала.

Хотя вы можете объявить всю эту информацию явно в каждом проектном объекте или системной архитектуре , часто более легким является объявление системной информации в отдельном блоке объявлений. Каждый объект системы может впоследствии использовать системный блок объявлений.

Объявление блока имеет следующий синтаксис :

package ***package_name*** **is**
{ package_declarative_item }
end ***[package_name]*** **;**

где ***package_name*** - имя данного блока объявлений. Элемент ***package_declarative_item*** может быть любым из следующих :

- предложение **use** (для включения других блоков)
- объявление типа
- объявление подтипа
- объявление константы
- объявление сигнала
- объявление подпрограммы
- объявление компонента

В примере 3-5 приведены некоторые объявления блоков.

Пример 3-5. Простые объявления блока

```
package EXAMPLE is

        type BYTE is range 0 to 255;
        subtype NIBBLE is BYTE range 0 to 15;

        constant BYTE_FF: BYTE := 255;

        signal ADDEND: NIBBLE;

        component BYTE_ADDER
                port(A, B:          in BYTE;
                     C:             out BYTE;
                     OVERFLOW:      out BOOLEAN);
        end component;

        function MY_FUNCTION (A: in BYTE) return BYTE;

end EXAMPLE;
```

Для использования приведенного выше примера объявлений добавьте оператор **use** в начале описания вашего проекта следующим образом :

```
use WORK.EXAMPLE.ALL;
```

entity . . .

architecture . . .

Прочие примеры блоков и их объявлений приведены в блоках объявлений , обеспечиваемых Synopsys. Эти блоки перечислены в Приложении B.

Тела блока объявлений

В телах блока объявлений содержатся конкретные исполнения подпрограмм , перечисленных в объявлении блока. Тем не менее , эта информация никогда не видна проектам или объектам , которые используют данный блок. Тела блока могут включать исполнения (тела) подпрограмм , объявленных в объявлении блока и во внутренне поддерживаемых подпрограммах.

Тело блока объявлений имеет следующий синтаксис :

package body ***package_name*** **is**
{ package_body_declarative_item }
end ***[package_name]*** **;**

где ***package_name*** - имя соответствующего блока объявлений. Элемент ***package_body_declarative_item*** может быть любым из следующих :

- предложение **use**
- объявление подпрограммы
- тело подпрограммы
- объявление типа
- объявление подтипа
- объявление константы

Примеры объявлений и тел блоков вы можете увидеть в блоке **std_logic_arith** , поддерживаемым FPGA Express. Этот блок объявлений приведен в Приложении B.

Определение проектов

Высокоуровневые конструкции , обсужденные выше в данной главе , включают в себя :

- объектные спецификации (интерфейсы)
- объектные архитектуры (исполнения)
- подпрограммы

Объектные спецификации

Объектная спецификация определяет характеристики объекта , которые должны быть известны до того , как этот объект будет связан с другими объектами и компонентами. Например , перед тем , как вы соедините счетчик с другими объектами , вы должны определить количество и типы его входов и выходов. Объектная спецификация определяет порты (входы и выходы) объекта и имеет следующий синтаксис :

entity ***entity_name*** **is**
[**generic(** ***generic_declarations*****) ;** ***]***
[**port(** ***port_declarations*****) ;** ***]***
end ***[entity_name]*** **;**

где ***entity_name*** - имя объекта , ***generic_declarations*** определяет локальные константы , используемые для установления размеров или синхронизации объекта , а ***port_declarations*** определяет количество и тип входов и выходов.

Другие объявления не поддерживаются в объектной спецификации.

Основные (Generics) объектные спецификации

Основные спецификации являются параметрами объекта. Generics могут определять битовую ширину компонентов (таких как сумматоры) или внутренние временные значения. Основные спецификации могут иметь значение по умолчанию. Значение не по умолчанию назначается только в тех случаях , когда объект реализован (см. «Оператор реализации компонента» далее в этой главе) или сконфигурирован (см. «Объектные конфигурации» далее в этой главе). Внутри объекта основные спецификации являются константами. Синтаксис объявлений ***generic_declarations*** следующий :

generic(
[***constant_name*** **:** ***type*** **[:= *value*]**
{ ; ***constant_name*** **:** ***type*** **[:= *value*] }**
);

где ***constant_name*** - имя константы generic , ***type*** - ранее определенный тип данных , а необязательный параметр ***value*** является значением ***constant_name*** по умолчанию.

Примечание : FPGA Express поддерживает только целые (INTEGER) типы generics.

Спецификации портов объекта

Синтаксис объявлений ***port_declarations*** следующий :

port(
[***port_name*** **:** ***mode port_type***
{ ; ***port_name*** **:** ***mode port_type*****}]**
);

где ***port_name*** - имя порта ; ***mode*** - **in** , **out** , **inout** или **buffer** ; а ***port_type*** - ранее определенный тип данных. Существуют четыре режима работы порта :

in
Может осуществлять только чтение.

Out
Может только присваивать значение.

inout
Может осуществлять чтение и присваивать значение. Читаемое значение - это то , которое приходит на порт , а не то , которое ему назначается (если таковое имеется).

buffer
Аналогично режиму out , но может быть прочитан. Читаемое значение равно назначенному. Имеет только один источник (драйвер). Более подробную информацию о драйверах см. в разделе «Управляемые сигналы» в Главе 7.

В примере 3-6 показана объектная спецификация для 2-входового N-битового компаратора с битовой шириной по умолчанию равной 8.

Пример 3-6. Интерфейс для N-битового компаратора.

```
-- Определяем объект (проект) , называемый COMP ,
-- который имеет два N-битовых входа и один выход.

entity COMP is
        generic(N:      INTEGER := 8);          -- по умолчанию 8 бит

        port(X, Y:      in BIT_VECTOR (0 to N-1);
             EQUAL:     out BOOLEAN);
end COMP;
```

Каждая объектная архитектура определяет одно исполнение функции объекта. Архитектуру можно классифицировать в диапазоне от алгоритма (набор последовательных операторов внутри процесса) до структурного списка цепей (набор компонентных реализаций). Архитектура имеет следующий синтаксис :

architecture ***architecture_name*** **of** ***entity_name*** **is**
    { block_declarative_item }
begin
    { concurrent_statement }
end ***[architecture_name]*** **;**

где ***architecture_name*** - имя архитектуры , а ***entity_name*** - имя реализуемого объекта. Элемент ***block_declarative_item*** может быть любым из следующих :

- предложение **use**
- объявление подпрограммы
- тело подпрограммы
- объявление типа
- объявление подтипа
- объявление константы
- объявление сигнала
- объявление компонента

Параллельные операторы (***concurrent_statement***) описаны в главе 7.

В примере 3-7 показано завершенное описание схемы трехбитового счетчика , объектная спецификация (**COUNTER3**) и архитектура (**MY_ARCH**). Этот пример также включает результирующую синтезированную схему.

Пример 3-7. Реализация трехбитового счетчика.

```
entity COUNTER3 is
port (  CLK :           in bit;
        RESET:          in bit;
        COUNT:          out integer range 0 to 7);
end COUNTER3;

architecture MY_ARCH of COUNTER3 is
signal COUNT_tmp : integer range 0 to 7;
begin
        process
        begin
                wait until (CLK'event and CLK = '1');
                                        -- ожидаем синхроимпульс
                if RESET = '1' or COUNT_tmp = 7 then
                                        -- Ck. для сброса или макс. числа
                        COUNT_tmp <= 0;
                else COUNT_tmp <= COUNT_tmp + 1;
                                        -- продолжаем считать
                end if;
        end process;
        COUNT <= COUNT_tmp;
end MY_ARCH;
```

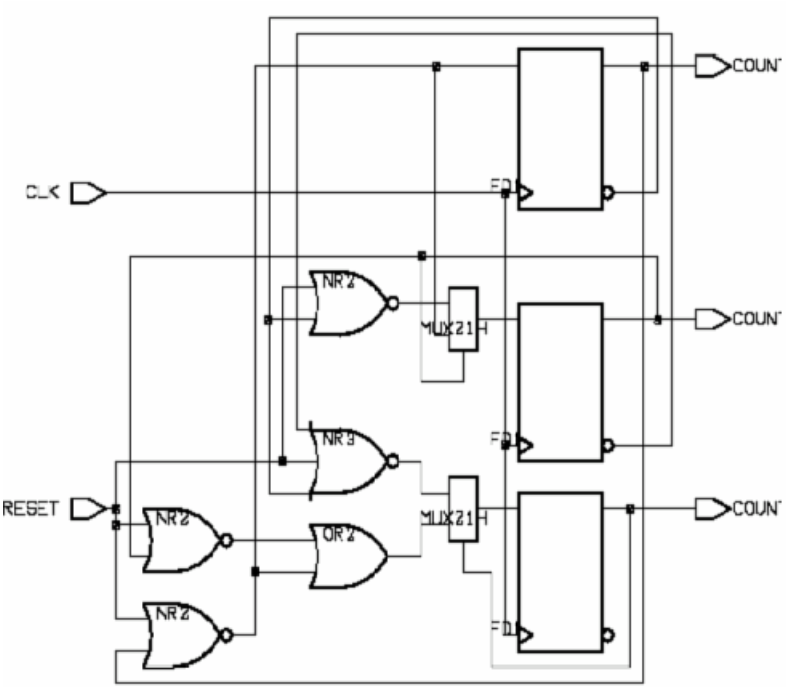


Рис.3-5. Схема трехбитового счетчика.

Примечание : В архитектуре вы не должны объявлять константы или сигналы под теми же именами , что и порты объекта. Если вы объявляете константу или сигнал с именем порта , то новое объявление «спрячет» имя порта. Если новое объявление включено в объявление архитектуры , (как показано в примере 3-8) , а не во внутренний блок , FPGA Express даст отчет об ошибке.

Пример 3-8. Некорректное использование имени порта при объявлении сигналов или констант.

```
entity X is
        port(SIG, CONST: in BIT;
             OUT1, OUT2: out BIT);
end X;

architecture EXAMPLE of X is
        signal SIG : BIT;
        constant CONST: BIT := '1';
begin

...

end EXAMPLE;
```

Сообщения об ошибках , генерируемые для примера 3-8 :

```
signal SIG : BIT;
       ^
Error: (VHDL-1872) line 13
        Illegal redeclaration of SIG. (Некорректное переобъявление SIG).

constant CONST: BIT := '1';
          ^
Error: (VHDL-1872) line 14
        Illegal redeclaration of CONST. (Некорректное переобъявление CONST).
```

Объектные конфигурации

Конфигурация определяет одну комбинацию объекта и архитектуры для проекта.

Примечание : FPGA Express поддерживает только конфигурации , которые соответствуют одному объекту верхнего уровня с архитектурой.

Поддерживаемый синтаксис конфигурации следующий :

configuration *configuration_name* of *entity_name* is
    for *architecture_name*
    end for;
end *[configuration_name]* ;

где ***configuration_name*** - имя данной конфигурации , ***entity_name*** - имя высокоуровневого объекта , а ***architecture_name*** - имя архитектуры , используемой для ***entity_name***. В примере 3-9 показана конфигурация для трехбитового счетчика из примера 3-7. Эта конфигурация соответствует объектной спецификации счетчика (**COUNTER3**) с архитектурой (**MY_ARCH**).

Пример 3-9. Конфигурация счетчика из примера 3-7.

```
configuration MY_CONFIG of COUNTER3 is
    for MY_ARCH
    end for;
end MY_CONFIG;
```

Примечание : Если вы не определяете конфигурацию для объекта с несколькими архитектурами , IEEE VHDL считает , что используется последняя прочитанная архитектура. Это определяется из файла .mra (самого последнего из проанализированных).

Подпрограммы

Подпрограммы описывают алгоритмы , которые предполагается использовать более , чем в одном проекте. В отличие от операторов компонентных реализаций , когда подпрограмма используется объектом или другой подпрограммой , новый уровень иерархии проекта не создается автоматически. Тем не менее вы можете вручную определить подпрограмму как новый уровень проектной иерархии в окне исполнения FPGA Express (Implementation Window).

Два типа подпрограмм , процедуры и функции , могут содержать ноль или более параметров :

процедуры
: Процедуры возвращают не значение , а информацию вызывающим операторам об изменении значений их собственных параметров.

функции
: Функция имеет одиночное значение , которое возвращается вызывающему оператору , однако она не может изменить значения этих параметров.

Так же , как и объект , подпрограмма состоит из двух частей - объявления и тела :

объявление
: Объявляет интерфейс к подпрограмме : ее имя , параметры и возвращаемое значение (если таковое имеется).

тело
: Определяет алгоритм , который обеспечивает ожидаемый от подпрограммы результат.

Когда вы объявляете подпрограмму в блоке объявлений , это объявление должно находиться в объявлении блока , а тело подпрограммы должно находиться в теле блока объявлений. Подпрограмма, объявляемая внутри архитектуры , имеет тело , но не имеет соответствующего объявления подпрограммы.

Объявления подпрограмм

В объявлении подпрограммы перечисляются имена и типы ее параметров , а для функций - тип возвращаемого значения.

Синтаксис объявления процедуры имеет следующий вид :

procedure ***proc_name*** **[(*parameter_declarations*)] ;**

где ***proc_name*** - имя процедуры.

Синтаксис объявления функции имеет следующий вид :

function ***func_name*** **[(*parameter_declarations*)]**
return ***type_name*** **;**

где ***func_name*** - имя функции , а ***type_name*** - тип возвращаемого функцией значения. Синтаксис ***parameter_declarations*** такой же , как и ***port_declarations*** :

[***parameter_name*** **:** ***mode parameter_type***
{ ; ***parameter_name*** **:** ***mode parameter_type*****}]**

где ***parameter_name*** - имя параметра ; ***mode*** - **in** , **out** , **inout** или **buffer** ; ***parameter_type*** - ранее определенный тип данных.

Параметры процедуры могут использовать любой режим. Параметры функции должны использовать только режим **in**. Сигнальные параметры диапазонного типа не могут быть пропущены через подпрограмму. В примере 3-10 показаны образцы объявлений для функции и процедуры.

Пример 3-10. Два объявления подпрограммы.

```
type BYTE is array (7 downto 0) of BIT;
type NIBBLE is array (3 downto 0) of BIT;

function IS_EVEN(NUM: in INTEGER) return BOOLEAN;
        -- Возвращает TRUE , если NUM четный.

procedure BYTE_TO_NIBBLES(B:                    in BYTE;
                          UPPER, LOWER:         out NIBBLE);
        -- Разбивает BYTE на две половины : UPPER и LOWER.
```

Примечание : Когда вы вызываете подпрограмму , действительные параметры заменяются объявленными формальными параметрами. Действительные параметры либо являются постоянной величиной , либо именем сигнала , переменной , константы или порта. Действительный параметр должен поддерживать тип и режим формального параметра. Например , входной порт не может быть использован как выходной действительный параметр , а константа может использоваться только в качестве входного действительного параметра.

В примере 3-11 показаны некоторые вызовы подпрограмм , объявленных в примере 3-10.

Пример 3-11. Два вызова подпрограммы.

```
signal INT : INTEGER;
variable EVEN : BOOLEAN;
. . .
INT <= 7;
EVEN := IS_EVEN(INT);
. . .

variable TOP, BOT: NIBBLE;
. . .
BYTE_TO_NIBBLES("00101101", TOP, BOT);
```

Тела подпрограмм

Тело подпрограммы определяет выполнение ее алгоритма. Синтаксис тела процедуры имеет следующий вид :

procedure ***procedure_name*** **[(*parameter_declarations*)**
] is
{ *subprogram_declarative_item* }
begin
{ *sequential_statement* }
end [*procedure_name*] ;

Синтаксис тела функции имеет следующий вид :

function ***function_name*** **[(*parameter_declarations*)]**
return *type_name* is
{ *subprogram_declarative_item* }
begin
{ *sequential_statement* }
end [*function_name*] ;

Элемент ***subprogram_declarative_item*** может быть любым из следующих :

- предложение **use**
- объявление типа
- объявление подтипа
- объявление константы
- объявление переменной
- объявление атрибута
- спецификация атрибута
- объявление подпрограммы
- тело подпрограммы

В примере 3-12 приведены тела подпрограмм , объявленных в примере 3-10.

Пример 3-12. Два тела подпрограмм.

```
function IS_EVEN(NUM: in INTEGER)
        return BOOLEAN is
begin
        return ((NUM rem 2) = 0);
end IS_EVEN;

procedure BYTE_TO_NIBBLES(B:                    in BYTE;
                          UPPER, LOWER:         out NIBBLE) is
begin
        UPPER := NIBBLE(B(7 downto 4));
        LOWER := NIBBLE(B(3 downto 0));
end BYTE_TO_NIBBLES;
```

Перегрузка подпрограмм

Вы можете перегрузить подпрограммы ; более , чем одна подпрограмма может носить одинаковое имя. Каждая подпрограмма , которая использует данное имя , должна иметь различный профиль параметров. Последний определяет номер подпрограммы и тип параметров. Эта информация указывает на то , какая из подпрограмм вызывается в том случае , если несколько из них имеют одинаковые имена. Перегружаемые функции , кроме того , отличаются типом возвращаемых значений. В примере 3-13 показаны две подпрограммы с одинаковыми именами , но различными профилями параметров.

Пример 3-13. Перегрузка подпрограмм.

```
type SMALL is range 0 to 100;
```

```
type LARGE is range 0 to 10000;

function IS_ODD(NUM: SMALL) return BOOLEAN;
function IS_ODD(NUM: LARGE) return BOOLEAN;

signal A_NUMBER: SMALL;
signal B: BOOLEAN;
. . .
B <= IS_ODD(A_NUMBER); -- Вызывает первую из описанных выше функций
```

Перегрузка операторов

Предопределенные операторы , такие как + , **and** и **mod** также могут быть перегружены. С помощью перегрузки вы можете адаптировать предопределенные операторы для работы с вашими собственными типами данных. Например , вы можете объявить новые логические типы вместо того , чтобы использовать предопределенные **BIT** и **INTEGER** . Однако , вы не можете использовать предопределенные операторы с этими новыми типами , пока для них не будут объявлены перегруженные операторы. В примере 3-14 показано , как некоторые предопределенные операторы перегружаются для нового логического типа.

Пример 3-14. Перегрузка операторов.

```
type NEW_BIT is ('0', '1', 'X');
        -- Новый логический тип

function "and"(I1, I2: in NEW_BIT) return NEW_BIT;
function "or" (I1, I2: in NEW_BIT) return NEW_BIT;
        -- Объявляем перегруженные операторы для нового логического типа
. . .
signal A, B, C: NEW_BIT;
. . .

C <= (A and B) or C;
```

При объявлении перегруженного типа VHDL требует заключать его имя или символ в двойные кавычки , поскольку в данном случае он является инфиксным оператором (т.е. используется между операндами). Если вы объявите перегруженные операторы без двойных кавычек , компилятор VHDL воспримет их как функции раньше , чем как операторы.

Объявления типов

Объявления типов определяют имя и характеристики типа. Типы и их объявления полностью описаны в Главе 4. Тип - это именованный набор значений , таких как набор целых или набор **(red, green, blue)** . Объект данного типа , такой как сигнал , может принимать любые имеющиеся в нем значения. В примере 3-14 показано объявление типа **NEW_BIT** , а также некоторых функций и переменных этого типа.

Объявления типов разрешены в архитектурах , блоках объявлений , объектах , блоках , процессах и подпрограммах.

Объявления подтипов

Используйте объявления подтипов для обозначения имени и характеристик ограниченного подмножества другого типа или подтипа. Подтип является полностью совместимым с родительским типом , но только в определенном диапазоне. Объявления подтипов описаны в Главе 4. В следующем объявлении подтипа (**NEW_LOGIC**) определяется поддиапазон типа , объявленного в примере 3-14.

```
subtype NEW_LOGIC is NEW_BIT range '0' to '1';
```

Объявления подтипов разрешены там же , где и объявления подтипов : в архитектурах , блоках объявлений , объектах , блоках , процессах и подпрограммах.

Объявления констант

Объявления констант создают именованные значения данного типа. Значение константы может быть прочитано , но не изменено. Объявления констант разрешены в архитектурах , блоках объявлений , объектах , блоках , процессах и подпрограммах. В примере 3-15 показаны некоторые объявления констант.

Пример 3-15. Объявления констант.

```
constant WIDTH      : INTEGER := 8;
constant X          : NEW_BIT := 'X';
```

Вы можете использовать константы в выражениях , как описано в Главе 5 , а также в качестве исходных значений в операторах присваивания , как описано в Главе 6.

Объявления сигналов

Объявления сигналов создают новые именованные сигналы (проводники) данного типа. Сигналам могут быть присвоены начальные значения (по умолчанию). Однако , эти начальные значения не используются для синтеза. Сигналы с несколькими источниками (сигналы , управляемые присоединенной логикой) могут иметь соответствующие функции разрешения , как описано в следующем разделе. В примере 3-16 показано два объявления сигналов.

Пример 3-16. Объявления сигналов.

```
signal  A, B    : BIT;
signal  INIT    : INTEGER := -1;
```

Примечание : Порты также являются сигналами , у которых есть ограничения : выходные порты не могут быть прочитаны , а входным не может быть присвоено значение. Сигналы создаются либо через объявления портов , либо через объявления сигналов. Порты создаются только через объявления портов.

Вы не можете объявлять сигналы в архитектурах , объектах и блоках , а также использовать их в процессах и подпрограммах. Процессы и подпрограммы не могут объявлять сигналы для внутреннего использования.

Вы можете использовать сигналы в выражениях , как описано в Главе 5. Значения назначаются сигналам с помощью операторов присвоения , описанных в Главе 6.

Функции разрешения

Функции разрешения используются с сигналами , которые могут быть связаны (соединены вместе). Например , если к сигналу непосредственно подсоединены два источника (драйвера) , функция разрешения устанавливает , является ли значение сигнала функцией И , ИЛИ или функцией третьего состояния от значений источников. Для моделирования вы можете записать произвольную функцию , чтобы избежать шинных конфликтов.

Примечание : Функция разрешения может изменить значение разрешаемого сигнала , даже если все источники имеют одинаковое значение.

Функция разрешения сигнала является частью объявления подтипа этого сигнала. Разрешаемый сигнал создается за четыре шага :

```
-- Шаг 1
type SIGNAL_TYPE is ...
-- базовый тип сигнала SIGNAL_TYPE

-- Шаг 2
subtype res_type is res_function SIGNAL_TYPE;
-- имя подтипа res_type
-- имя функции res_function
-- тип сигнала res_type (подтип от SIGNAL_TYPE)
...
-- Шаг 3
function res_function (DATA: ARRAY_TYPE)
return SIGNAL_TYPE is
-- объявление функции разрешения
-- ARRAY_TYPE должен быть безусловным подмножеством SIGNAL_TYPE
...
-- Шаг 4
signal resolved_signal_name:res_type;
-- resolved_signal_name - разрешаемый сигнал
...
```

1. Объявляется базовый тип сигнала.
2. Подтип разрешаемого сигнала объявляется как подтип базового типа и включает имя функции разрешения.
3. Объявление самой функции разрешения (и далее ее определение).
4. Разрешаемые сигналы объявлены как разрешаемые подтипы.

FPGA Express не поддерживает произвольные функции разрешения. Разрешены только AND , OR и функция третьего состояния. FPGA Express требует , чтобы вы помечали все функции разрешения специальной директивой , показывающей тип выполняемого разрешения.

Примечание : FPGA Express учитывает директивы только при окончательной разводке аппаратного обеспечения. Тело функции разрешения анализируется , но игнорируется. Использование не поддерживаемых конструкций VHDL (см. Приложение С) приводит к генерации ошибки. Не соединяйте сигналы , которые используют различные функции разрешения. FPGA Express поддерживает только одну функцию разрешения на цепь.

Существует три директивы функций разрешения :

```
-- synopsys resolution_method wired_and
-- synopsys resolution_method wired_or
-- synopsys resolution_method three_state
```

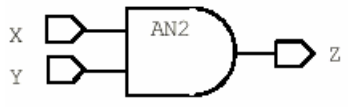
Примечание : Результаты моделирования до и после синтеза могут не согласовываться между собой , если тело функции разрешения , используемой симулятором , не совпадает с директивой , используемой синтезатором.

В примере 3-17 показано , как создать и использовать разрешаемые сигналы , а также применение директив компилятора для функций разрешения. Базовым типом сигнала является предопределенный тип **BIT**.

Пример 3-17. Разрешаемый сигнал и его функция разрешения.

```
package RES_PACK is
        function RES_FUNC(DATA: in BIT_VECTOR) return BIT;
        subtype RESOLVED_BIT is RES_FUNC BIT;
end;

package body RES_PACK is
        function RES_FUNC(DATA: in BIT_VECTOR) return BIT
is
```

```
        -- pragma resolution_method wired_and
begin
        -- Код данной функции игнорируется FPGA ,
Express
        -- однако анализируется на предмет корректного синтаксиса VHDL
                for I in DATA'range loop
                        if DATA(I) = '0' then
                                return '0';
                        end if;
                end loop;
                return '1';
        end;
end;

use work.RES_PACK.all;

entity WAND_VHDL is
        port(X, Y: in BIT; Z: out RESOLVED_BIT);
end WAND_VHDL;

architecture WAND_VHDL of WAND_VHDL is
begin
        Z <= X;
        Z <= Y;
end WAND_VHDL;
```



Объявления переменных

Объявления переменных определяют именованное значение данного типа. Вы можете использовать переменные в выражениях , как описано в Главе 5. Переменным присваиваются значения с помощью операторов присвоения , описанных в главе 6. В примере 3-18 показаны некоторые объявления переменных.

Пример 3-18. Объявления переменных.

```
variable A, B: BIT;
variable INIT: NEW_BIT;
```

Примечание : Переменные объявляются и используются только в процессах и подпрограммах , так как процессы и подпрограммы не могут объявлять сигналы для внутреннего использования.

Структурное проектирование

FPGA Express работает с одним или несколькими проектами. Каждый объект (и архитектура) в описании VHDL транслируется в одиночный проект в FPGA Express. Кроме того , в оригиналах проекты могут находиться в отличных от VHDL форматах , таких как уравнения , программируемые логические матрицы (PLA) , машинные состояния , другие языки HDL или списки цепей. Проект может содержать низкоуровневые объекты , соединенные цепями (сигналами) с низкоуровневыми портами проекта. Такие низкоуровневые проекты могут содержать и другие объекты из проектов VHDL , проектов , представленных в каких-либо других форматах Synopsys или ячеек из технологической библиотеки. Реализуя проекты внутри проектов , вы создаете иерархию.

Иерархия в VHDL определяется через использование объявлений компонентов и операторов компонентной реализации. Для включения проекта вы должны определить его интерфейс вместе с

объявлением компонентов. Затем вы можете создать реализацию данного проекта с помощью *оператора* компонентной реализации. Если ваш проект содержит только объекты VHDL , то каждый оператор объявления компонента соответствует объекту в проекте. Если ваш проект использует другие проекты или ячейки из технологической библиотеки , не описанные в VHDL , создаются объявления компонентов без соответствующих объектов. Затем вы можете использовать FPGA Express для связи VHDL компонента с не-VHDL проектом или ячейкой.

Примечание : Для моделирования вашего проекта VHDL вы должны обеспечить описания объявлений объектов и архитектур для всех компонентных объявлений.

Использование компонентов аппаратного уровня

VHDL включает конструкции для использования существующих аппаратных компонентов. Такие структурные конструкции могут быть использованы для определения списка цепей компонентов. В следующих разделах описано , как использовать компоненты , и как FPGA Express конфигурирует эти компоненты.

Компонентные объявления

Вы должны объявить компонент в архитектуре или блоке объявлений до того , как использовать (реализовать) его. Оператор компонентного объявления аналогичен оператору объектной спецификации , описанному ранее , в котором определяется компонентный интерфейс. Компонентное объявление имеет следующий синтаксис :

```
component identifier
        [ generic( generic_declarations ) ]
        [ port( port_declarations ) ]
end component ;
```

где ***identifier*** - имя данного типа компонента , а синтаксис элементов ***generic_declarations*** и ***port_declarations*** такой же , как у определенных ранее для объектных спецификаций. В примере 3-19 показан простой оператор компонентного объявления.

Пример 3-19. Компонентное объявление двухвходового элемента И.

```
component AND2
        port(I1, I2:    in BIT;
             O1:        out BIT);
end component;
```

В примере 3-20 показан оператор компонентного объявления , который использует общий (generic) параметр.

Пример 3-20. Компонентное объявление N-битового сумматора.

```
component ADD
        generic(N: POSITIVE);

        port(   X, Y:           in BIT_VECTOR(N-1 downto 0);
                Z:              out BIT_VECTOR(N-1 downto 0);
                CARRY:          out BIT)
end component;
```

Хотя оператор компонентного объявления аналогичен объектной спецификации , он служит другой цели. Компонентное объявление требуется тля того , чтобы объект **AND2** или **ADD** стал применимым , или видимым , внутри архитектуры. После объявления компонента он может быть использован в проекте.

Источники компонентов

Объявленный компонент может находиться в том же файле описания VHDL , другом файле VHDL , другом формате (например , формате обмена электронными данными - EDIF) или таблице состояний , а также в технологической библиотеке. Если компонента нет ни в одном из текущих исходных файлов VHDL , он должен быть уже откомпилирован FPGA Express. Когда проект , который использует компоненты , компилируется FPGA Express , то предварительно откомпилированные компоненты ищутся по имени в следующем порядке :

1. В текущем проекте.
2. Во входном исходном файле или файлах , идентифицированных в окне исполнения FPGA Express.
3. В библиотеках специфических технологических компонентов FPGA.

Совместимость портов компонента

FPGA Express проверяет совместимость объектов VHDL. Для других объектов имена портов берутся из оригинального описания проекта.

- Для компонентов из технологической библиотеки именами портов являются имена входных и выходных выводов.
- Для проектов EDIF именами портов являются имена портов EDIF.

Битовая ширина каждого порта также должна быть согласована. FPGA Express проверяет совместимость для компонентов VHDL , потому что типы портов должны быть идентичными. Для компонентов из других источников FPGA Express осуществляет проверку при линковании компонента в описание VHDL.

Оператор реализации компонента

Оператор компонентной реализации исполняет и связывает компоненты для формирования структурного описания (списка цепей) проекта. Оператор реализации компонента может создавать новый уровень иерархии проекта , он имеет следующий синтаксис :

```
instance_name : component_name
[ generic map (
        generic_name => expression
        { , generic_name => expression }
) ]
port map (
        [ port_name => ] expression
        { , [ port_name => ] expression }
);
```

где ***instance_name*** - имя реализации компонента типа ***component_name***. Необязательный элемент **generic map** назначает иные , чем по умолчанию , общие значения. Каждый элемент ***generic_name*** является именем generic , точно так же , как указывалось в соответствующем операторе объявления компонента. Каждое выражение ***expression*** представляет соответствующее значение. **port map** назначает порты для связей компонента. Каждое имя ***port_name*** является именем порта , точно так же, как указывалось в соответствующем операторе объявления компонента. Каждое выражение ***expression*** представляет значение сигнала.

FPGA Express использует следующие два правила для решения , какой объект и архитектуру ассоциировать с реализацией компонента :

1. Каждое объявление компонента должно иметь объект с таким же именем : объект VHDL , проект из другого источника (в другом формате) или библиотеку компонентов. Этот объект используется для каждой компонентной реализации , соответствующей объявлению компонента.
2. Если объект VHDL имеет более , чем одну архитектуру , то для каждой компонентной реализации , соответствующей объявлению компонента , используется *последняя* введенная архитектура. Файл **.mra** определяет последнюю проанализированную архитектуру.

Размещение значений Generic

При реализации компонента с общими значениями (generics) вы можете присвоить им определенные значения. Если у generic нет значения по умолчанию , то они должны реализовываться с параметром **genericmap**. Например , четырехбитовая реализация компонента **ADD** из примера 3-20 должна использовать следующий **genericmap** :

```
U1: ADD genericmap (N => 4)
        port map (X, Y, Z, CARRY...);
```

port map назначает порты компонента действительным сигналам ; это описано в следующем разделе.

Размещение связей портов

Вы можете определить связи портов в операторах компонентной реализации как при помощи нужного поименования , так и при помощи соответствующего расположения. При использовании именованной записи конструкция ***port_name*** **=>** идентифицирует определенные порты компонента. При использовании позиционной записи выражения для портов компонента просто перечисляются в порядке объявления. В примере 3-21 показана именованная и позиционная запись оператора компонентной реализации для **U5**.

Пример 3-21. Эквивалентные именованная и позиционная записи.

```
U5: or2 port map (O => n6, I1 => n3, I2 => n1);
        -- Именованная запись
U5: or2 port map (n3, n1, n6);
        -- Позиционная запись
```

Примечание : Когда вы используете позиционную запись , то реализуемые выражения (сигналы) для порта должны следовать в том же порядке , как при объявлении портов.

В примере 3-22 показано структурное описание (список цепей) для объекта **COUNTER3** из примера 3-7.

Пример 3-22. Структурное описание трехбитового счетчика.

```
architecture STRUCTURE of COUNTER3 is
        component DFF
                port(CLK , DATA: in BIT;
                                Q: out BIT);
        end component;
        component AND2
                port(I1, I2: in BIT;
                        O: out BIT);
        end component;
        component OR2
                port(I1, I2: in BIT;
                        O: out BIT);
        end component;
        component NAND2
                port(I1, I2: in BIT;
                        O: out BIT);
        end component;
        component XNOR2
                port(I1, I2: in BIT;
                        O: out BIT);
        end component;
        component INV
```

```
            port(I: in BIT;
                 O: out BIT);
        end component;

        signal N1, N2, N3, N4, N5, N6, N7, N8, N9: BIT;

begin
        u1: DFF port map(CLK, N1, N2);
        u2: DFF port map(CLK, N5, N3);
        u3: DFF port map(CLK, N9, N4);
        u4: INV port map(N2, N1);
        u5: OR2 port map(N3, N1, N6);
        u6: NAND2 port map(N1, N3, N7);
        u7: NAND2 port map(N6, N7, N5);
        u8: XNOR2 port map(N8, N4, N9);
        u9: NAND2 port map(N2, N3, N8);
        COUNT(0) <= N2;
        COUNT(1) <= N3;
        COUNT(2) <= N4;
end STRUCTURE;
```

Технологически независимая реализация компонента

При использовании структурного стиля проектирования вы можете захотеть реализовать многие логические компоненты. Для этих целей Synopsys обеспечивает общую технологическую библиотеку **GTECH**. В этой библиотеке содержатся технологически независимые логические компоненты , такие как :

- элементы И , ИЛИ и исключающее ИЛИ (2 , 3 , 4 , 5 и 8 входов)
- однобитовые сумматоры и полусумматоры
- схемы приоритета 2-из-3
- мультиплексоры
- триггера и защелки
- многоуровневые логические элементы , такие как И-НЕ , И-ИЛИ , И-ИЛИ-НЕ

Вы можете использовать эти простые компоненты для создания технологически независимых проектов. В примере 3-23 показано , как N-битовый сумматор с переносом может быть создан из N однобитовых сумматоров.

Пример 3-23. Проект , использующий технологически независимые компоненты.

```
library GTECH;
use gtech.gtech_components.all;
entity RIPPLE_CARRY is
        generic(N: NATURAL);

        port(   A, B:           in BIT_VECTOR(N-1 downto 0);
                CARRY_IN:       in BIT;
                SUM:            out BIT_VECTOR(N-1 downto 0);
                CARRY_OUT: out BIT;);
end RIPPLE_CARRY;

architecture TECH_INDEP of RIPPLE_CARRY is

signal CARRY: BIT_VECTOR(N downto 0);

begin
        CARRY(0) <= CARRY_IN;
```

```
        GEN: for I in 0 to N-1 generate
                U1: GTECH_ADD_ABC port map(
                A => A(I),
                B => B(I),
                C => CARRY(I),
                S => SUM(I),
                COUT => CARRY(I+1));

        end generate GEN;

        CARRY_OUT <= CARRY(N);
end TECH_INDEP;
```

Глава 4. Типы данных

VHDL является строго типизированным языком. Каждая константа , сигнал , переменная , функция и параметр объявляются с определенным типом , таким как **BOOLEAN** или **INTEGER** , и могут хранить или возвращать значение только этого типа.

VHDL предопределяет абстрактные типы данных , такие как **BOOLEAN** , которые являются частью большинства языков программирования , и аппаратно связанные типы , таки как **BIT** , имеющиеся в большинстве аппаратных языков. Предопределенные типы VHDL объявлены в блоке объявлений **STANDARD** , который поддерживается всеми исполнениями VHDL (см. пример 4-12). Типы данных адресуют информацию о :

- перечисляемых типах
- целых типах
- типах массивов
- типах записи
- предопределенных типах данных VHDL
- неподдерживаемых типах данных
- типах данных Synopsys
- подтипах

Преимуществом строгой типизации является то , что средства VHDL могут отследить наиболее общие ошибки проекта , такие как назначение восьмибитового значения четырехбитовому сигналу или инкрементация индекса массива за пределы его диапазона. В следующем фрагменте программы показано обозначение нового типа , **BYTE** , как массива из восьми бит , и объявление переменной **ADDEND** , которая использует этот тип.

```
type BYTE is array(7 downto 0) of BIT;
variable ADDEND: BYTE;
```

Предопределенные типы данных VHDL выстроены из основных типов данных VHDL. Некоторые типы VHDL не поддерживаются для синтеза , такие как **REAL** и **FILE** . Примеры в данной главе показывают обозначения типов и соответствующие объявления объектов. Хотя каждая константа , сигнал , переменная , функция и параметр объявляются с определенным типом , здесь приведены примеры только для объявлений переменных и сигналов. Объявления констант , функций и параметров показаны в главе 3.

VHDL , кроме того , обеспечивает *подтипы* , которые определяются как подмножества других типов. В любом месте , где может появиться определение типа , может появиться и определение подтипа. Разница между типом и подтипом заключается в том , что подтип является подмножеством ранее определенного родительского (или базового) типа или подтипа. Перекрывающиеся подтипы данного базового типа могут быть сравнены и назначены друг другу. Все целые типы , например , являются техническими подтипами встроенного базового целого типа (см. «Целые типы» позже в этой главе). Подтипы описаны в последнем разделе данной главы.

Перечисляемые типы

Перечисляемый тип определяется списком (перечислением) всех возможных значений этого типа. Перечисляемые типы имеют следующий синтаксис определения :

type *type_name* is (*enumeration_literal*
{ , *enumeration_literal* });

где ***type_name*** - идентификатор типа , а каждый ***enumeration_literal*** - либо идентификатор (***enum_6)*** , либо литерал символа ***('A').*** Идентификатор является последовательностью букв , символов подчеркивания и цифр. Идентификатор должен начинаться с буквы и не может являться зарезервированным словом VHDL , таким как TYPE . Все зарезервированные слова VHDL перечислены в Приложении C. Литерал символа является любым значением типа CHARACTER в одиночных кавычках. В примере 4-1 показаны два определения перечисляемых типов и соответствующие объявления сигналов и переменных.

Пример 4-1. Определения перечисляемого типа.

```
type COLOR is (BLUE, GREEN, YELLOW, RED);
type MY_LOGIC is ('0', '1', 'U', 'Z');
variable HUE: COLOR;
signal SIG: MY_LOGIC;
. . .
HUE := BLUE;
SIG <= 'Z';
```

Перегрузка перечисления

Вы можете перегрузить перечисляемый литерал путем включения его в определение двух или более перечисляемых типов. Если вы используете такие перегруженные перечисляемые литералы , FPGA Express обычно может определить его тип. Тем не менее , при определенных обстоятельствах такое определение может быть невозможным. В этом случае вы должны квалифицировать литерал путем явного установления его типа (см. «Квалифицированные выражения» в Главе 5). В примере 4-2 показано , как вы можете квалифицировать перегруженный перечисляемый литерал.

Пример 4-2. Перегрузка перечисляемого литерала.

```
type COLOR is (RED, GREEN, YELLOW, BLUE, VIOLET);
type PRIMARY_COLOR is (RED, YELLOW, BLUE);
...
A <= COLOR'(RED);
```

Кодирование перечисления

Перечисляемые типы упорядочиваются путем перечисления *значений*. По умолчанию первому перечисляемому литералу присваивается значение **0** , следующему - **1** и т.д. FPGA Express автоматически кодирует перечисляемые значения в битовые вектора , которые основаны на каждой позиции значения. Длина кодирующего битового вектора равна минимальному количеству бит , необходимых для кодирования номера перечисляемых значений. Например , перечисляемый тип с пятью значениями имеет трехбитовый кодирующий вектор. В примере 4-3 показано кодирование по умолчанию перечисляемого типа с пятью значениями.

Пример 4-3. Автоматическое кодирование перечисления.

```
type COLOR is (RED, GREEN, YELLOW, BLUE, VIOLET);
```

Перечисляемые значения будут закодированы следующим образом :
RED ⇒ "000"
GREEN ⇒ "001"

YELLOW ⇒ "010"
BLUE ⇒ "011"
VIOLET ⇒ "100"

Результат равен **RED** < **GREEN** < **YELLOW** < **BLUE** < **VIOLET.**

Вы можете отменить автоматическое кодирование перечисления и определить свои собственные коды с помощью атрибута **ENUM_ENCODING**. Такая интерпретация является специфической для FPGA Express. Атрибут VHDL определяется своим именем и типом , и объявляется со значением для типа атрибута , как показано ниже в примере 4-4.

Примечание : Несколько атрибутов VHDL , связанных с синтезом , объявлены в блоке объявлений ATTRIBUTES , поддерживаемым FPGA Express. Этот блок приведен в Приложении B. Раздел «Атрибуты и ограничения синтеза» в Главе 9 описывает , как использовать эти атрибуты VHDL.

Атрибут **ENUM_ENCODING** должен быть строкой (**STRING)** , содержащей набор векторов , по одному для каждого перечисляемого литерала в соответствующем типе. Кодирующий вектор определяется символами **'0'** , **'1'** , **'D'** , **'U'** и **'Z'** , разделенными пробелами. Значение кодирующих векторов описано в следующем разделе. Первый вектор в строке атрибута определяет код для первого перечисляемого литерала , второй вектор - для второго литерала и т.д. Атрибут **ENUM_ENCODING** должен следовать сразу же за объявлением типа. Пример 4-4 иллюстрирует , как кодирование по умолчанию из примера 4-3 может быть изменено с помощью атрибута **ENUM_ENCODING**.

Пример 4-4. Использование атрибута ENUM_ENCODING .

```
attribute ENUM_ENCODING: STRING;
-- Определение атрибута

type COLOR is (RED, GREEN, YELLOW, BLUE, VIOLET);
attribute ENUM_ENCODING of
COLOR: type is "010 000 011 100 001";
-- Объявление атрибута
```

Перечисляемые значения будут закодированы следующим образом :
RED = "010"
GREEN = "000"
YELLOW = "011"
BLUE = "100"
VIOLET = "001"
Результат равен **GREEN<VIOLET<RED<YELLOW<BLUE**

Примечание : Интерпретация атрибута ENUM_ENCODING специфична для FPGA Express. Другие средства VHDL , такие как симуляторы , используют стандартное кодирование (стандартный порядок перечисления).

Значения кодов перечисления

Возможные значения кодов перечисления для атрибута **ENUM_ENCODING** следующие :

- **'0'** значение бита **0.**
- **'1'** значение бита **1.**
- **'D'** не важно (может быть либо **0** , либо **1**).
- **'U'** неизвестно. Если **U** появляется в кодирующем векторе перечисления , то вы не можете использовать этот перечисляемый литерал как операнд в операторах = и /= . Вы можете прочитать перечисляемый литерал , закодированный с **U** , из переменной или сигнала , но не можете назначить его. При синтезе оператор = возвратит **FALSE** , а оператор /= возвра-

тит **TRUE** , если любой из операндов , являющихся перечисляемым литералом , закодирован как **U**.

- **'Z'** Высокий импеданс. См. «Реализация третьего состояния» в Главе 8 для более подробной информации.

Целые типы

Максимальный диапазон чисел целого типа VHDL от **-(2^{32})** до **2^{32}** (**-2_147_483_647 .. 2_147_483_647**). Целые типы определяются как поддиапазоны этого анонимного встроенного типа. Многоцифровые числа в VHDL могут включать символы подчеркивания (_) для лучшей читаемости. FPGA Express кодирует целое значение как битовый вектор , длина которого является минимально необходимой для хранения определенного диапазона чисел , включая отрицательные в виде двоичного дополнения. Определение целого типа имеет следующий синтаксис :

type *type_name* is range *integer_range* ;

где ***type_name*** - имя нового целого типа , а ***integer_range*** - поддиапазон анонимного целого типа. В примере 4-5 показаны некоторые определения целого типа.

Пример 4-5. Определения целого типа.

```
type PERCENT is range -100 to 100;
-- Представляется как 8-битовый вектор
-- (1 знаковый бит , 7 бит значения)

type INTEGER is range -2147483647 to 2147483647;
-- Представляется как 32-битовый вектор
-- Это определение типа INTEGER
```

Примечание : Вы не можете непосредственно обращаться к битам INTEGER или явно указывать битовую ширину типа. По этой причине Synopsys обеспечивает перегружаемые функции для арифметических операций. Эти функции определены в блоке объявлений std_logic , приведенном в Приложении B.

Типы массивов

Массив является объектом , в котором собраны элементы одного типа. VHDL поддерживает N-размерные массивы , но FPGA Express поддерживает только одноразмерные массивы. Элементы массива могут быть любого типа. У массива есть индекс , значение которого отмечает каждый элемент. Диапазон индекса определяет , как много элементов находится в массиве , а также их порядок (снизу вверх или сверху вниз). Индекс может быть из любого целого типа. Вы можете объявить многоразмерные массивы , построив одноразмерный массив , тип элементов которого будет другим одноразмерным массивом , как показано в примере 4-6.

Пример 4-6. Объявление массива массивов

```
type BYTE is array (7 downto 0) of BIT;
type VECTOR is array (3 downto 0) of BYTE;
```

VHDL поддерживает как ограниченные , так и неограниченные массивы. Разница между ними заключается в диапазоне индекса при объявлении массива.

Ограниченный массив

Диапазон индекса ограниченного массива определен явно ; например , целый диапазон **(1 to 4)** . Когда вы объявляете переменную или сигнал такого типа , то он получает такой же диапазон индекса. Определение типа ограниченного массива имеет следующий синтаксис :

type ***array_type_name*** **is**
array (***integer_range*** **) of** ***type_name*** **;**

где ***array_type_name*** - имя нового типа ограниченного массива , ***integer_range*** - поддиапазон другого целого типа , а ***type_name*** - тип каждого элемента массива. В примере 4-7 показано определение ограниченного массива.

Пример 4-7. Определение типа ограниченного массива.

```
type BYTE is array (7 downto 0) of BIT;
-- Ограниченный массив , диапазон индекса которого равен
-- (7, 6, 5, 4, 3, 2, 1, 0)
```

Неограниченный массив

Диапазон индекса неограниченного массива определяется как *тип* , например , **INTEGER** . Такое определение подразумевает , что диапазон индекса может состоять из любого непрерывного подмножества значений этого типа. Когда вы объявляете переменную или сигнал данного типа , вы также определяете и действительный диапазон индекса. Различные определения могут иметь различные диапазоны индексов. Определение типа неограниченного массива имеет следующий синтаксис :

type ***array_type_name*** **is**
array (*range_type_name* **range <>)**
of ***element_type_name*** **;**

где ***array_type_name*** - имя нового типа неограниченного массива , ***range_type_name*** - имя целого типа или подтипа , а ***element_type_name*** - тип каждого элемента массива. В примере 4-8 показано определение типа неограниченного массива и объявление , которое использует его.

Пример 4-8. Определение типа неограниченного массива

```
type BIT_VECTOR is array(INTEGER range <>) of BIT;
-- Определение неограниченного массива
. . .
variable MY_VECTOR : BIT_VECTOR(5 downto -5);
```

Преимуществом использования неограниченных массивов является то , что средства VHDL запоминают диапазон индекса каждого объявления. Вы можете использовать *атрибуты массива* для установления диапазона (границ) сигнала или переменной типа неограниченный массив. Пользуясь этой информацией , вы можете писать программы , которые используют переменные или сигналы типа неограниченный массив , независимо от размера границ переменной или сигнала. В следующем разделе описываются атрибуты массива и способы их применения.

Атрибуты массива

FPGA Express поддерживает следующие предопределенные атрибуты VHDL для использования с массивами :

- **left** (влево)
- **right** (вправо)
- **high** (верхний)
- **low** (нижний)
- **length** (длина)

- **range** (диапазон)
- **reverse_range** (обратный диапазон)

Эти атрибуты возвращают значение соответствующей части диапазона массива. В табл.4-1 показаны значения атрибутов массива для переменной **MY_VECTOR** из примера 4-8.

Таблица 4-1. Атрибуты индекса массива.

MY_VECTOR'left	5
MY_VECTOR'right	**-5**
MY_VECTOR'high	**5**
MY_VECTOR'low	**5**
MY_VECTOR'length	**11**
MY_VECTOR'range	**(5 down to -5)**
MY_VECTOR'reverse_range	**(-5 to 5)**

В примере 4-9 показано использование атрибутов массива в функции , которая делает ИЛИ всех элементов данного **BIT_VECTOR** (объявленного в примере 4-8) и возвращает это значение.

Пример 4-9. Использование атрибутов массива.

```
function OR_ALL (X: in BIT_VECTOR) return BIT is
        variable OR_BIT: BIT;
        begin
                OR_BIT := '0';
                for I in X'range loop
                        OR_BIT := OR_BIT or X(I);
                end loop;

                return OR_BIT;
        end;
```

Заметим , что эта функция работает с **BIT_VECTOR** любого размера.

Типы запись

Запись является набором именованных полей различных типов (в противоположность массиву) , который составлен из идентичных анонимных вводов. Поле записи может быть любого ранее определенного типа , включая другой тип запись.

Примечание : Константы типа запись не поддерживаются в VHDL для синтеза (не поддерживается инициализация записей).

В примере 4-11 показано объявление типа запись (**BYTE_AND_IX**) , три сигнала этого типа и некоторые назначения.

Пример 4-11. Объявление и использование типа запись.

```
constant LEN: INTEGER := 8;

subtype BYTE_VEC is BIT_VECTOR(LEN-1 downto 0);

type BYTE_AND_IX is
        record
                BYTE: BYTE_VEC;
                IX: INTEGER range 0 to LEN;
        end record;
```

```
signal X, Y, Z: BYTE_AND_IX;

signal DATA: BYTE_VEC;
signal NUM: INTEGER;
. . .
X.BYTE <= "11110000";
X.IX <= 2;

DATA <= Y.BYTE;
NUM <= Y.IX;

Z <= X;
```

Как показано в примере 4-11 , вы можете читать значения из записи или назначать ей значения следующими способами :

- через индивидуальное имя поля
  ```
  X.BYTE <= DATA;
  X.IX <= LEN;
  ```
- из другого объекта записи того же типа
  ```
  Z <= X;
  ```

Примечание : Обращение к индивидуальным полям объекта типа запись осуществляется через имя объекта , точку и имя поля : X.BYTE или X.IX. Для обращения к элементу массивного поля BYTE используйте обозначение массива X.BYTE(2).

Предопределенные типы данных VHDL

IEEE VHDL описывает два специфических блока объявлений , каждый из которых содержит стандартный набор типов и операций : **STANDARD** и **TEXTIO**. Блок объявлений типов данных **STANDARD** включается во все исходные файлы VHDL через неявное предложение **use**. Блок **TEXTIO** определяет типы и операции для связи со стандартной средой программирования (терминал и файлы ввода/вывода). Этот блок объявлений не нужен для синтеза , то есть FPGA Express его не поддерживает.

Исполнение FPGA Express блока объявлений **STANDARD** приведено в примере 4-12. Этот блок **STANDARD** является подмножеством блока IEEE VHDL **STANDARD**. Отличия между ними описаны ниже в разделе «Неподдерживаемые типы данных».

Пример 4-12. Блок объявлений FPGA Express STANDARD.

```
package STANDARD is
        type BOOLEAN is (FALSE, TRUE);
        type BIT is ('0', '1');
        type CHARACTER is (
                NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
                BS, HT, LF, VT, FF, CR, SO, SI,
                DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
                CAN, EM, SUB, ESC, FSP, GSP, RSP, USP,
                ' ', '!', '"', '#', '$', '%', '&', ''',
                '(', ')', '*', '+', ',', '-', '.', '/',
                '0', '1', '2', '3', '4', '5', '6', '7',
                '8', '9', ':', ';', '<', '=', '>', '?',
                '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
                'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
                'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
                'X', 'Y', 'Z', '[', '\', ']', '^', '_',
                '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
```

```
                'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
                'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
                'x', 'y', 'z', '{', '|', '}', '~', DEL);
        type INTEGER is range -2147483647 to 2147483647;
        subtype NATURAL is INTEGER range 0 to 2147483647;
        subtype POSITIVE is INTEGER range 1 to 2147483647;
        type STRING is array (POSITIVE range <>)
                of CHARACTER;
        type BIT_VECTOR is array (NATURAL range <>)
                of BIT;
end STANDARD;
```

Тип данных *BOOLEAN*

Тип данных **BOOLEAN** в действительности является перечисляемым типом с двумя значениями , **FALSE** и **TRUE** , где **FALSE** < **TRUE** . Логические функции , такие как равенство (=) и сравнение (<) возвращают значение **BOOLEAN**.

Преобразование значения **BIT** в **BOOLEAN** выглядит следующим образом :

```
BOOLEAN_VAR := (BIT_VAR = '1');
```

Тип данных *BIT*

Тип данных **BIT** представляет двоичное значение одного из двух символов , **'0'** или **'1'**. Логические операции , таки как **and** , могут принимать и возвращать значения **BIT**.

Преобразование значения **BOOLEAN** в **BIT** выглядит следующим образом :

```
if (BOOLEAN_VAR) then
        BIT_VAR := '1';
else
        BIT_VAR := '0';
end if;
```

Тип данных *CHARACTER*

Тип данных **CHARACTER** перечисляет набор символов ASCII. Непечатаемые символы представлены трехбуквенными именами , такими как **NUL** для нулевого символа. Печатаемые символы представлены самими собой в одиночных кавычках следующим образом :

```
variable CHARACTER_VAR: CHARACTER;
. . .
CHARACTER_VAR := 'A';
```

Тип данных *INTEGER*

Тип данных **INTEGER** представляет положительные и отрицательные целые числа и ноль.

Тип данных *NATURAL*

Тип данных **NATURAL** является подмножеством **INTEGER** , которое используется для представления натуральных (неотрицательных) чисел.

Тип данных *POSITIVE*

Тип данных **POSITIVE** является подмножеством **INTEGER** , которое используется для представления положительных (ненулевых и неотрицательных) чисел.

Тип данных *STRING*

Тип данных **STRING** является неограниченным массивом типа **CHARACTER**. Значение **STRING** заключается в двойные кавычки следующим образом :

```
variable STRING_VAR: STRING(1 to 7);
. . .
STRING_VAR := "Rosebud";
```

Тип данных BIT_VECTOR

Тип данных **BIT_VECTOR** представляет массив значений **BIT**.

Неподдерживаемые типы данных

Некоторые типы данных являются либо непригодными для синтеза , либо не поддерживаются. Неподдерживаемые типы анализируются , но игнорируются FPGA Express. Эти типы перечислены и описаны ниже.

В приложении C описан уровень поддержки FPGA Express для каждой конструкции VHDL.

Физические типы

FPGA Express не поддерживает физические типы , такие как единицы измерений (например , **nS**). Поскольку физические типы относятся к процессу моделирования , FPGA Express разрешает , но игнорирует их объявление.

Типы с плавающей точкой

FPGA Express не поддерживает типы с плавающей точкой , такие как **REAL** . *Литералы* с плавающей точкой , такие как **1.34** , разрешены в определениях FPGA Express-поддерживаемых атрибутов.

Типы доступа

FPGA Express не поддерживает типы доступа (векторные типы) , поскольку для них не существует эквивалентных аппаратных конструкций.

Файловые типы

FPGA Express не поддерживает файловые типы (дисковые файлы). Аппаратным файлом является ОЗУ или ПЗУ.

Типы данных SYNOPSYS

Блок объявлений **std_logic_arith** обеспечивает арифметические операции и числовые сравнения над данными типа массив. Блок также определяет два основных типа данных : **UNSIGNED** и **SIGNED** . Эти типы данных , в отличие от предопределенного типа **INTEGER** , обеспечивают доступ к индивидуальным битам (проводникам) численного значения. Более подробную информацию вы найдете в Приложении B.

Подтипы

Подтип определяется как подмножество ранее определенного типа или подтипа. Определение подтипа может появиться в любом месте , где разрешено определение типа. Подтипы являются мощным способом использования проверки типов VHDL для уверенности в правильности назначений и обработки данных. Подтипы наследуют все операторы и подпрограммы , определенные для родительских (базовых) типов.

Подтипы также используются для разрешающих сигналов , соответствующих функции разрешения. (См. «Объявления сигналов» в Главе 3). Например , в примере 4-12 **NATURAL** и **POSITIVE** являются подтипами **INTEGER** , и могут использоваться с любой функцией типа **INTEGER**. Эти подтипы могут добавляться , перемножаться , сравниваться и назначаться друг другу, пока значения находятся внутри соответствующего диапазона подтипа. Все типы и подтипы **INTEGER** в действительности являются подтипами анонимного предопределенного численного типа. В примере 4-13 показаны некоторые правильные и неправильные назначения между **NATURAL** и **POSITIVE**.

Пример 4-13. Правильные и неправильные назначения между целыми подтипами.

```
variable NAT: NATURAL;
variable POS: POSITIVE;
. . .
POS := 5;
NAT := POS + 2;
. . .
NAT := 0;
POS := NAT; -- Неправильно ; за пределами диапазона
Например , тип BIT_VECTOR определяется как
type BIT_VECTOR is array(NATURAL range <>) of BIT;
```

Если ваш проект использует только 16-битовые вектора , вы можете определить подтип **MY_VECTOR** как

```
subtype MY_VECTOR is BIT_VECTOR(0 to 15);
```

В примере 4-14 показано , что все функции и атрибуты , которые работают с **BIT_VECTOR** , также работают и с **MY_VECTOR** .

Пример 4-14. Атрибуты и функции , работающие с подтипом.

```
type BIT_VECTOR is array(NATURAL range <>) of BIT;
subtype MY_VECTOR is BIT_VECTOR(0 to 15);
. . .
signal VEC1, VEC2: MY_VECTOR;
signal S_BIT: BIT;
variable UPPER_BOUND: INTEGER;
. . .
if (VEC1 = VEC2)
. . .
VEC1(4) <= S_BIT;
VEC2 <= "0000111100001111";
. . .
RIGHT_INDEX := VEC1'high;
```

Глава 5. Выражения

Выражения выполняют арифметические и логические вычисления путем назначения оператора одному или более операндам. Операторы определяют тип выполняемых вычислений. Операнды являются данными для осуществления вычислений. Выражения обсуждаются как

- операторы
- операнды

В следующем фрагменте VHDL **A** и **B** являются операндами , + является оператором , а **A + B** является выражением.

```
C := A + B; -- Вычисляет сумму двух значений
```

Вы можете использовать выражения в различных местах описания проекта. Выражения могут осуществлять :

- Назначения переменным или сигналам , либо используются как начальные значения констант.
- Использоваться как операнды других операторов.
- Использоваться для возвращения значений функций.
- Использоваться в качестве входных параметров при вызове подпрограммы.
- Назначаться выходным параметрам в теле процедуры.

- Использоваться для управления работой операторов , таких как **if** , **loop** и **case** .

Для лучшего понимания выражений VHDL обсудим индивидуальные компоненты операторов и операндов.

Операторы

- Логические операторы
- Операторы сравнения
- Операторы сложения
- Унарные (знаковые) операторы
- Операторы умножения
- Смешанные арифметические операторы

Операнды

- Вычисляемые операнды
- Литералы
- Идентификаторы
- Индексные имена
- Секторные имена
- Множества
- Атрибуты
- Вызовы функций
- Ограниченные выражения
- Преобразования типов

Операторы

Оператор VHDL характеризуется

- именем
- функцией вычисления
- количеством операндов
- типом операндов (таким , как **Boolean** или **Character**)
- типом возвращаемого значения

Вы можете определить новые операторы , такие как функции , для любого типа операндов и возвращаемого значения. Предопределенные операторы VHDL перечислены в табл.5-1. Каждая строка таблицы представляет операторы одного приоритета. Приоритет операторов возрастает сверху вниз. В соответствии с приоритетом определяется порядок выполнения смежных операторов в выражении.

Таблица 5-1. Предопределенные операторы VHDL .

Тип	**Операторы**					**Приоритет**
Логический	**and**	**or**	**nand**	**nor**	**xor**	Низший
Сравнение	=	/=	<	<= >	>=	
Сложение	+	-	**&**			
Унарный (знак)	+	-				
Умножение	*	/	**mod**	**rem**		
Смешанный	**	**abs**	**not**			Высший

В примере 5-1 показаны некоторые выражения и их интерпретация.

Пример 5-1. Приоритет операторов

A + B * C = A + (B * C)
not BOOL and (NUM = 4) = (not BOOL) and (NUM = 4)

VHDL позволяет перегружать существующие операторы (назначать им новые типы операндов). Например , оператор **and** может быть перегружен для работы с новым логическим типом. Более подробную информацию см. в разделе «Перегрузка операторов» Главы 3.

Логические операторы

Операнды логического оператора должны быть того же типа. Логические операторы **and , or , nand , nor , xor** и **not** принимают операнды типа **BIT , BOOLEAN** и одноразмерные массивы типа **BIT** или **BOOLEAN** . Операнды типа массив должны быть того же размера. Логический оператор , назначенный двум массивам , воздействует на пары элементов этих массивов. В примере 5-2 показаны некоторые объявления логических сигналов и логические операции над ними.

Пример 5-2. Логические операторы

```
signal A, B, C:         BIT_VECTOR(3 downto 0);
signal D, E, F, G:      BIT_VECTOR(1 downto 0);
signal H, I, J, K:      BIT;
signal L, M, N, O, P:   BOOLEAN;
A <= B and C;
D <= E or F or G;
H <= (I nand J) nand K;
L <= (M xor N) and (O xor P);
```

Обычно , чтобы использовать в выражении более двух операндов , вам необходимы скобки для их группирования. Вместо этого вы можете скомбинировать последовательность операторов **and** , **or** или **xor** без скобок , например , **A and B and C and D .** Однако , такие последовательности с разными операторами , например , **A or B xor C** все-таки требуют скобок для расстановки нужных приоритетов. В примере 5-3 используются объявления из примера 5-2 для иллюстрации некоторых общих ошибок.

Пример 5-3. Ошибки при использовании логических операторов.

```
H <= I and J or K;                -- Нужны скобки;
L <= M nand N nand O nand P;  -- Нужны скобки;
A <= B and E;                     -- Операнды должны быть одного размера;
H <= I or L;                      -- Операнды должны быть одного типа;
```

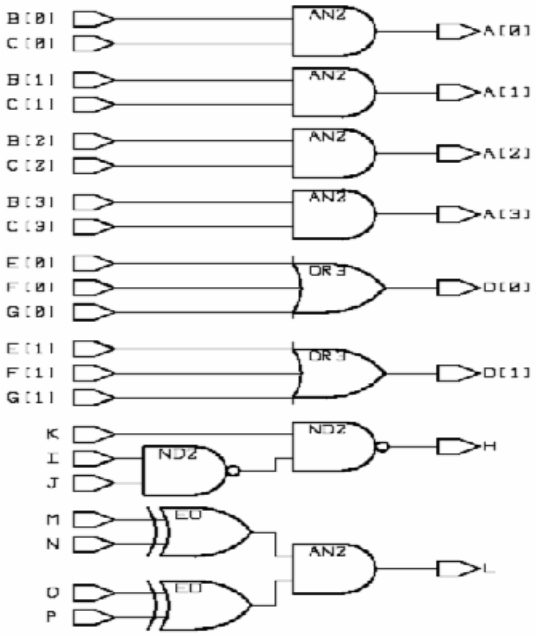


Рис.5-1. Общие ошибки при использовании логических операторов.

Операторы сравнения

Операторы сравнения , такие как = или > , сравнивают два операнда одного базового типа и возвращают значение **BOOLEAN**. IEEE VHDL определяет операторы равенства (=) и неравенства (/=) для всех типов. Два операнда равны , если они представляют одинаковое значение. Для типов массив и запись IEEE VHDL сравнивает соответствующие элементы операндов.

IEEE VHDL определяет порядковые операторы (< , <= , "" (оператор сравнения)">> , и =""(оператор сравнения)">>=) для всех перечисляемых типов , целых типов и одноразмерных массивов перечисляемого или целого типов. Внутренний порядок значений данного типа определяет результат порядковых операторов. Целые значения расположены от минус бесконечности до плюс бесконечности. Перечисляемые значения расположены в том порядке , в каком они были объявлены , пока вы не изменили кодирование.

Примечание : Если вы устанавливаете кодирование ваших перечисляемых типов (см. «Кодирование перечисления» в Главе 4) , то порядковые операторы сравнивают порядок ваших закодированных значений , но не порядок объявления. Поскольку такая интерпретация специфична для FPGA Express, симулятор VHDL продолжает использовать порядок объявления для перечисляемых типов.

Массивы упорядочены так же , как слова в словаре. Относительный порядок значений двух массивов определяется путем сравнения каждой пары элементов по очереди , начиная с левой границы диапазона индекса каждого массива. Если пара элементов не равна , то порядок отличающихся элементов определяет порядок массивов. Например , битовый вектор **101011** меньше , чем **1011** , так как четвертый бит каждого вектора отличается , а **0** меньше **1**.

Если два массива имеют разную длину , и короткий массив согласуется с первой частью длинного , то короткий массив упорядочен перед длинным. Таким образом , битовый вектор **101** меньше , чем **101000** . Массивы сравниваются слева направо , независимо от диапазонов индексов (**to** или **downto**). В примере 5-4 показано несколько выражений , дающих результат **TRUE** .

Пример 5-4. Истинные выражения отношения.

```
'1' = '1'
"101" = "101"
"1" > "011"      -- Сравнение массивов
"101" < "110"
```

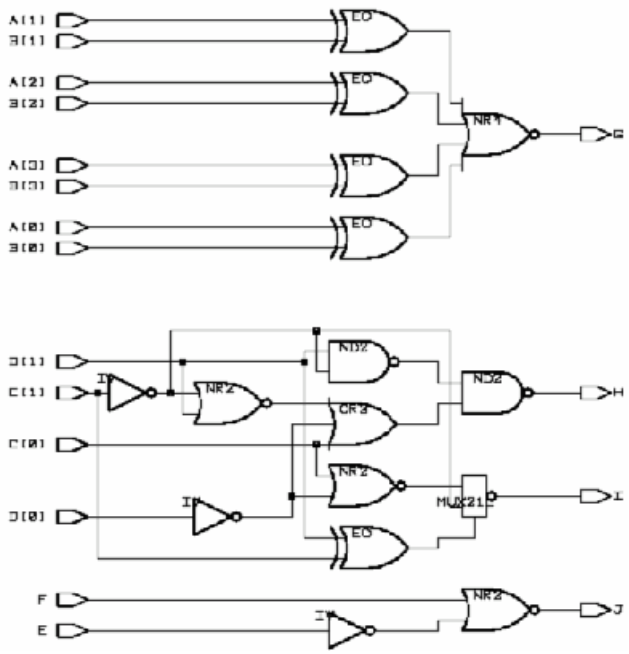
Для интерпретации битовых векторов , таких как **011** , как знаковых или беззнаковых двоичных чисел , используйте операторы отношения , определенные в блоке объявлений FPGA Express **std_logic_arith** (приведен в Приложении B). Третья строка в пример 5-4 даст результат **FALSE** , если операнды будут типа **UNSIGNED** .

UNSIGNED'"1" < UNSIGNED'"011" – Сравнение чисел

В примере 5-5 показаны некоторые выражения отношения и результирующие синтезированные схемы.

Пример 5-5. Операторы сравнения.

```
signal A, B: BIT_VECTOR(3 downto 0);
signal C, D: BIT_VECTOR(1 downto 0);
signal E, F, G, H, I, J: BOOLEAN;
G <= (A = B);
H <= (C < D);
I <= (C >= D);
J <= (E > F);
```



Операторы сложения

Операторы сложения включают *арифметические* операторы и операторы *конкатенации* (сочленения). Арифметические операторы + и - предопределены FPGA Express для всех целых операндов. Эти операторы сложения и вычитания выполняют стандартные арифметические действия, как показано в примере 5-6. Для сумматоров и вычитателей шириной более четырех бит используется библиотека синтетических компонентов (см Главу 9).

Оператор конкатенации (**&)** предопределен для всех одноразмерных массивов. Этот оператор выстраивает массивы путем комбинирования с их операндами. Каждый операнд **&** может быть массивом или элементом массива. Используйте **&** для добавления одиночного элемента в начало или конец массива , комбинирования двух массивов или для построения массива из элементов , как показано в примере 5-6.

Пример 5-6. Операторы сложения.

```
signal A, D:      BIT_VECTOR(3 downto 0);
signal B, C, G:   BIT_VECTOR(1 downto 0);
signal E:         BIT_VECTOR(2 downto 0);
```

signal F, H, I: BIT;

signal J, K, L: INTEGER range 0 to 3;

A <= not B & not C; -- Массив & массив
D <= not E & not F; -- Массив & элемент
G <= not H & not I; -- Элемент & элемент
J <= K + L; -- Простое сложение

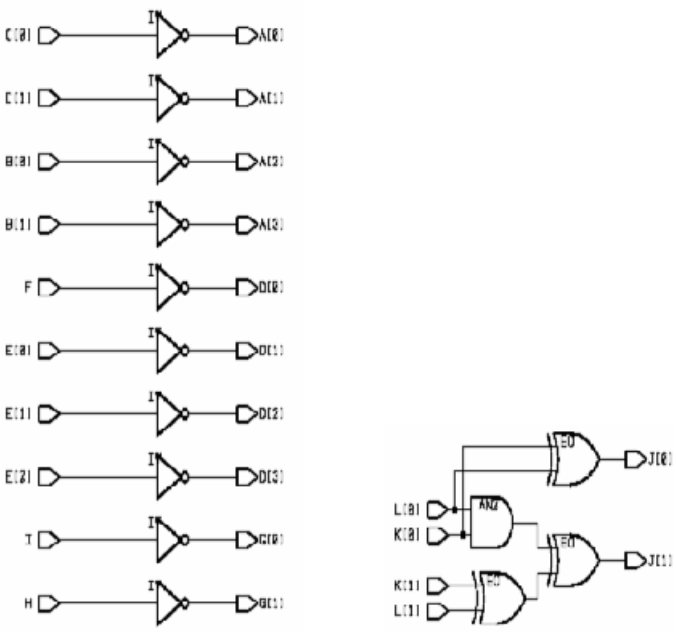


Рис.5-2. Операторы сложения.

Унарные (знаковые) операторы

Унарный оператор имеет только один операнд. FPGA Express предопределяет унарные операторы + и - для всех целых типов. Оператор + не оказывает никакого воздействия. Оператор - меняет знак своего операнда. Например ,

5 = +5
5 = -(-5)

В примере 5-7 показано , как синтезируется унарное отрицание.

Пример 5-7. Унарные (знаковые) операторы.

signal A, B: INTEGER range -8 to 7;
A <= -B;

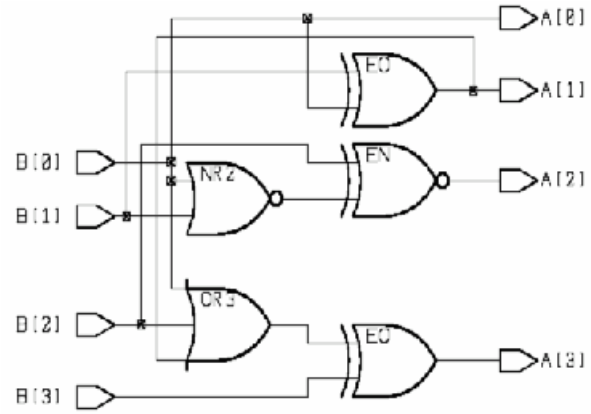


Рис.5-3. Унарные (знаковые) операторы.

Операторы умножения

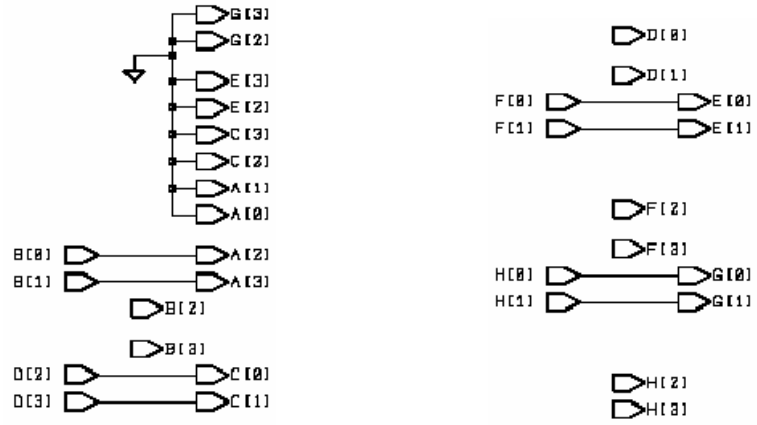
FPGA Express предопределяет операторы умножения (*****, /, **mod**, и **rem**) для всех целых типов. FPGA Express накладывает некоторые ограничения на поддерживаемые значения правых операндов операторов умножения следующим образом :

- * Целое умножение : нет ограничений. Оператор умножения выполняется как синтетическая библиотечная ячейка.
- / Целое деление : правый операнд должен быть *вычислимой* степенью 2 (см. «Вычисляемые операнды» позже в этой главе). Ни один из операндов не может быть отрицательным. Этот оператор выполняется в виде битового сдвига.
- **mod** Модуль : то же , что и для /.
- **rem** Остаток : то же , что и для /.

В примере 5-8 показаны некоторые использования операторов умножения , правые операнды которых являются степенью 2. Также показаны результирующие синтезированные схемы.

Пример 5-8. Операторы умножения со степенью 2

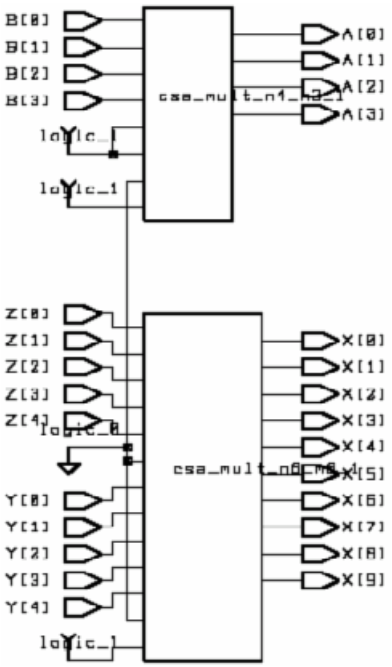
```
signal A, B, C, D, E, F, G, H: INTEGER range 0 to 15;

        A <= B * 4;
        C <= D / 4;
        E <= F mod 4;
        G <= H rem 4;
```



В примере 5-9 показаны две операции умножения , одна для четырехбитового операнда и двухбитовой константы (**B * 3**) , а другая - для двух пятибитовых операндов (**X * Y**). Поскольку синтетическая библиотека доступна по умолчанию , то эти умножения выполняются через ее ячейки.

Пример 5-9. Оператор умножения (*) , использующий синтетические ячейки.

```
signal A, B: INTEGER range 0 to 15;
signal Y, Z: INTEGER range 0 to 31;
signal X: INTEGER range 0 to 1023;
. . .
        A <= B * 3;
        X <= Y * Z;
```



Смешанные арифметические операторы

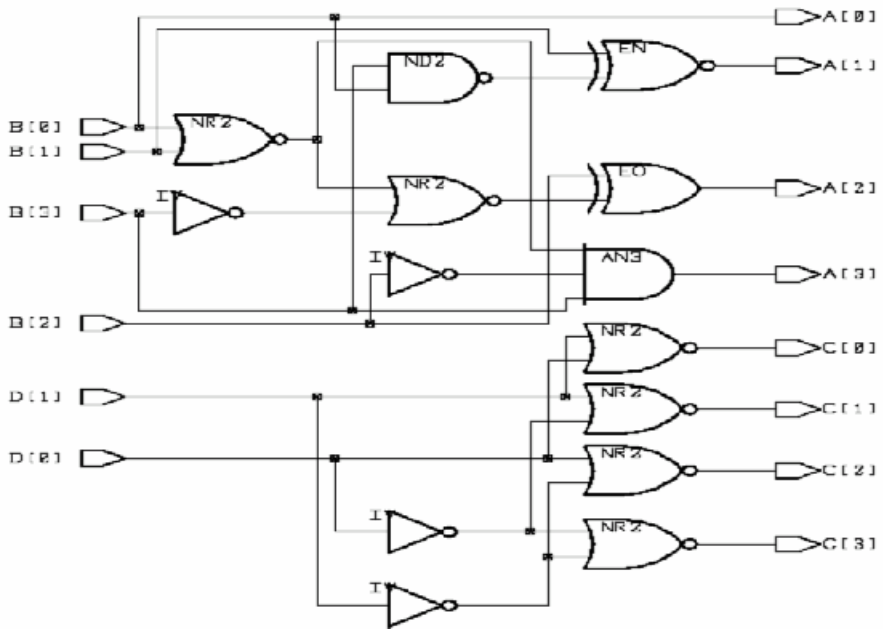
FPGA Express предопределяет операторы абсолютного значения (**abs**) и возведения в степень (**) для всех целых типов. FPGA Express устанавливает единственное ограничение для операции ** :

- ** Возведение в степень : Левый операнд должен быть вычислимой степенью 2 (см. «Вычисляемые операнды» позже в этой главе).

В примере 5-10 показано , как используются и синтезируются данные операторы.

Пример 5-10. Смешанные арифметические операторы.

```
signal A, B: INTEGER range -8 to 7;
signal C: INTEGER range 0 to 15;
signal D: INTEGER range 0 to 3;

    A <= abs(B);
    C <= 2 ** D;
```



Операнды

Операнды определяют данные , используемые оператором для вычисления его значения. Говорят , что операнд возвращает свое значение оператору. Существует много категорий операндов . Простейшим операндом является литерал , такой как число 7 , или идентификатор , такой как переменная или имя сигнала. Сам операнд может являться выражением. Вы создаете выражения-операнды путем заключения выражения в скобки.

Категории операндов следующие :

- Выражения : **(A nand B)**
- Литералы : **'0', "101", 435, 16#FF3E#**
- Идентификаторы : **my_var , my_sig**
- Индексные имена : **my_array(7)**
- Скользящие имена : **my_array(7 to 11)**
- Поля : **my_record.a_field**
- Множества : **my_array_type'(others => 1)**
- Атрибуты : **my_array'range**
- Вызовы функций : **LOOKUP_VAL(my_var_1, my_var_2)**
- Определенные выражения :**BIT_VECTOR'('1' & '0')**
- Преобразования типов : **THREE_STATE('0')**

В следующих двух разделах обсуждается битовая ширина операндов и объясняются вычисляемые операнды. Последующие разделы описывают типы операндов , перечисленные выше.

Битовая ширина операндов

FPGA Express использует битовую ширину наибольшего операнда для определения ширины , необходимой для аппаратной реализации оператора. Например , операнд **INTEGER** по умолчанию имеет размер 32 бита. Сложение двух операндов **INTEGER** заставит FPGA Express строить 32-битовый сумматор. Для эффективного использования аппаратных ресурсов всегда объявляйте битовый размер численных операндов. Например , пользуйтесь поддиапазоном **INTEGER** , при объявлении типов , переменных или сигналов.

```
type ENOUGH: INTEGER range 0 to 255;
variable WIDE: INTEGER range -1024 to 1023;
signal NARROW: INTEGER range 0 to 7;
```

Примечание : В процессе оптимизации FPGA Express удаляет аппаратные средства для неиспользуемых бит.

Вычисляемые операнды

Некоторые операторы , такие как оператор деления , ограничивают свои операнды , чтобы они были *вычисляемыми*. Вычисляемым называется такой операнд , значение которого может быть определено FPGA Express. Вычисляемость является важной , поскольку не вычисляемые выражения могут потребовать дополнительных логических элементов для определения их значения.

Ниже приведены примеры вычисляемых операндов :

- Значения литералов
- Параметры **for ... loop** , когда диапазон цикла является вычисляемым
- Переменные , которым назначено вычисляемое выражение
- Множества , которые содержат только вычисляемые выражения
- Вызовы функций с вычисляемым возвращаемым значением
- Выражения с вычисляемым операндом
- Определенные выражения , где выражение является вычисляемым
- Преобразования типов , когда выражение является вычисляемым
- Значение операторов **and** или **nand** , когда один из операндов является вычисляемым **0**

- Значение операторов **or** или **nor** , когда один из операндов является вычисляемой **1**

Кроме того , переменной присваивается вычисляемое значение , если есть **OUT** или **INOUT** параметр процедуры , который назначает ей вычисляемое значение.

В следующих примерах представлены не вычисляемые операнды :

- Сигналы
- Порты
- Переменные , которым присваиваются различные вычисляемые значения , зависящие от не вычисляемого условия
- Переменные , которым назначены не вычисляемые значения

В примере 5-11 показаны некоторые определения и объявления с последующими вычисляемыми и не вычисляемыми выражениями.

Пример 5-11. Вычисляемые и не вычисляемые выражения.

```
signal S: BIT;
. . .
function MUX(A, B, C: BIT) return BIT is
begin
        if (C = '1') then
                return(A);
        else
                return(B);
        end if;
end;

procedure COMP(A: BIT; B: out BIT) is
begin
        B := not A;
end;

process(S)
        variable V0, V1, V2: BIT;
        variable V_INT: INTEGER;

        subtype MY_ARRAY is BIT_VECTOR(0 to 3);
        variable V_ARRAY: MY_ARRAY;
begin
        V0 := '1';          -- Вычисляемое (значение равно '1')
        V1 := V0;           -- Вычисляемое (значение равно'1')
        V2 := not V1;       -- Вычисляемое (значение равно'0')

        for I in 0 to 3 loop
                V_INT := I;      -- Вычисляемое (значение зависит от итерации)
        end loop;

        V_ARRAY := MY_ARRAY'(V1, V2, '0', '0');          -- Вычисляемое ("1000")

        V1 := MUX(V0, V1, V2);          -- Вычисляемое (значение равно '1')
        COMP(V1, V2);
        V1 := V2;                       -- Вычисляемое (значение равно '0')
        V0 := S and '0';                -- Вычисляемое (значение равно '0')
        V1 := MUX(S, '1', '0');         -- Вычисляемое (значение равно '1')
        V1 := MUX('1', '1', S);         -- Вычисляемое (значение равно '1')

        if (S = '1') then
                V2 := '0';              -- Вычисляемое (значение равно '0')
        else
                V2 := '1';              -- Вычисляемое (значение равно '1')
```

```
        end if;
        V0 := V2;                    -- Не вычисляемое; V2 зависит от S
        V1 := S;                     -- Не вычисляемое; S является сигналом
        V2 := V1;                    -- Не вычисляемое; V1 больше не вычисляемое
end process;
```

Литералы

Литеральный операнд (константа) может быть численным , символьным , перечисляемым или строковым. В последующих разделах описываются эти четыре типа литералов.

Численные литералы

Численные литералы являются константами целого типа. Существует два типа численных литералов - десятичные и базовые. Десятичные литералы записываются по основанию 10. Базовые литералы могут быть записаны по основанию от 2 до 16 и состоят из основания , символа решетки (#), числа по данному основанию и еще одной решетки (#) ; например , **2#101#** равно десятичному 5. Цифры в любом из типов численных литералов могут разделяться символами подчеркивания (_). В примере 5-12 показано несколько разных численных литералов , представляющих одно и то же значение.

Пример 5-12. Численные литералы.

```
170
1_7_0
10#170#
2#1010_1010#
16#AA#
```

Символьные литералы

Символьные литералы являются одиночными символами , заключенными в кавычки , например , **A**. Символьные литералы могут использоваться как значения для операторов и для определения перечисляемых типов , таких как **CHARACTER** и **BIT**. См. Главу 4 для более подробной информации о разрешенных символьных типах.

Перечисляемые литералы

Перечисляемые литералы являются значениями перечисляемых типов. Существует два типа перечисляемых литералов - символьные литералы и идентификаторы. Символьные литералы были описаны раньше. Перечисляемые идентификаторы также являются литералами , перечисленными при определении перечисляемого типа. Например :

```
type SOME_ENUM is ( ENUM_ID_1, ENUM_ID_2, ENUM_ID_3);
```

Если два перечисляемых типа используют одни и те же литералы , то говорят , что они *перегружены*. Вы должны определять перегруженные перечисляемые литералы (см. «Определенные выражения» ниже в этой главе) , когда вы используете их в выражениях , если их тип не может быть определен из контекста. См. Главу 4 о более подробной информации. В примере 5-13 определены два перечисляемых типа и показаны некоторые значения перечисляемых литералов.

Пример 5-13. Перечисляемые литералы.

```
type ENUM_1 is (AAA, BBB, 'A', 'B', ZZZ);
type ENUM_2 is (CCC, DDD, 'C', 'D', ZZZ);

AAA                  -- Перечисляемый идентификатор типа ENUM_1
```

```
'B'              -- Символьный литерал типа ENUM_1
CCC              -- Перечисляемый идентификатор типа ENUM_2
'D'              -- Символьный литерал типа ENUM_2
ENUM_1'(ZZZ)     -- Определяется , так как перегружен
```

Строковые литералы

Строковые литералы являются одноразмерными массивами символов , заключенными в двойные кавычки (" "). Существует два типа строковых литералов - символьные строки и битовые строки. *Символьные строки* являются последовательностями символов в двойных кавычках ; например , **"ABCD"** . *Битовые строки* аналогичны символьным , однако представляют двоичные , восьмеричные или шестнадцатеричные значения ; например , **B"1101"** , **O"15"** и **X"D"** - все представляют десятичное значение 13.

Тип строкового значения является одноразмерным массивом перечисляемого типа. Каждый из символов строки представляет один элемент массива. В пример 5-14 показаны некоторые символьно-строковые литералы.

Пример 5-14. Символьно-строковые литералы.

```
"10101"
"ABCDEF"
```

Примечание : Нулевые строковые литералы ("") не поддерживаются.

Битовые строки так же , как численные литералы по основанию , состоят из *символа , определяющего основание* , двойной кавычки , последовательности чисел по данному основанию и второй двойной кавычки. Например , **B"0101"** представляет битовый вектор 0101. Битно-строковые литералы состоят из определителя основания **B** , **O** , или **X** , вслед за которым следует строковый литерал. Битно-строковый литерал интерпретируется как *битовый вектор* , одноразмерный массив предопределенного типа **BIT**. Определитель основания указывает на интерпретацию битовой строки следующим образом :

B *(двоичный)*
: Значение в виде двоичных цифр (*биты* , 0 или 1). Каждый бит в строке представляет один бит в генерируемом битовом векторе (массиве).

O *(восьмеричный)*
: Значение в виде восьмеричных цифр (от 0 до 7). Каждая восьмеричная цифра в строке представляет три бита в генерируемом битовом векторе (массиве).

X *(шестнадцатеричный)*
: Значение в виде шестнадцатеричных цифр (от 0 до 9 и от A до F). Каждая шестнадцатеричная цифра в строке представляет четыре бита в генерируемом битовом векторе (массиве).

Вы можете разделять цифры в битно-строковом литерале символами подчеркивания (_) для лучшей читаемости. В примере 5-15 показано несколько битно-строковых литералов , которые представляют одинаковое значение.

Пример 5-15. Битно-строковые литералы.

```
X"AAA"
B"1010_1010_1010"
O"5252"
B"101_010_101_010"
```

Идентификаторы

Идентификаторы , вероятно, являются наиболее общими операндами. Идентификатор является именем константы , переменной , сигнала , объекта , порта , подпрограммы или параметра и воз-

вращает значение объекта операнду. В пример 5-16 показаны различные типы идентификаторов и их использование. Все идентификаторы показаны жирным наклонным шрифтом.

Пример 5-16. Идентификаторы.

```
entity EXAMPLE is
        port (  INT_PORT: in INTEGER;
                BIT_PORT: out BIT);
end;
. . .
signal  BIT_SIG: BIT;
signal  INT_SIG: INTEGER;
. . .
INT_SIG <= INT_PORT;        -- Сигнал назначается из порта
port
BIT_PORT <= BIT_SIG;        -- Сигнал назначается порту

function FUNC(INT_PARAM: INTEGER)
        return INTEGER;
end function;
. . .
constant CONST:         INTEGER := 2;
variable VAR:           INTEGER;
. . .
VAR := FUNC(INT_PARAM => CONST);        -- Вызов функции
```

Индексные имена

Индексное имя идентифицирует один элемент массива переменной или сигнала. *Скользящие имена* идентифицируют последовательность элементов в массиве переменной или сигнала ; *множества* создают массивы литералов путем присвоения значения каждому элементу реализации типа массив. Скользящие имена и множества описаны в следующих двух разделах.

Индексное имя имеет следующий синтаксис :

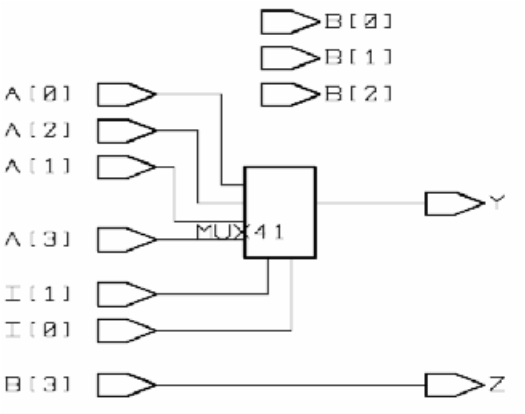
identifier **(** ***expression*** **)**

где ***identifier*** - имя сигнала или переменной типа массив. Выражение ***expression*** должно возвращать значение в пределах диапазона индекса массива. Значение , возвращаемое оператору , определяется элементом массива.

Если ***expression*** является вычисляемым (см. «Вычисляемые операнды» ранее в этой главе) , то операнд синтезируется непосредственно. Если выражение не вычисляемое , то синтезируется аппаратное обеспечение , которое распаковывает указанный элемент из массива. В примере 5-17 показано два индексных имени - одно вычисляемое и одно не вычисляемое.

Пример 5-17. Индексные имена в качестве операндов.

```
signal A, B:    BIT_VECTOR(0 to 3);
signal I:       INTEGER range 0 to 3;
signal Y, Z:    BIT;

Y <= A(I);      -- Не вычисляемое индексное выражение
Z <= B(3);      -- Вычисляемое индексное выражение
```



Вы также можете использовать индексные имена как назначаемые указатели ; см. «Индексные имена-указатели» в Главе 6.

Скользящие имена

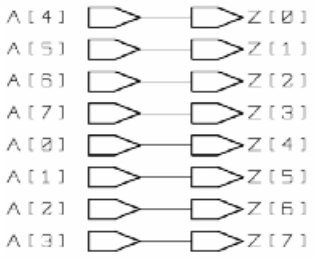
Скользящие имена возвращают последовательность элементов массива и имеют следующий синтаксис :

***identifier* (*expression direction expression*)**

где ***identifier*** - имя сигнала или переменной типа массив. Каждое выражение ***expression*** должно возвращать значение в диапазоне индекса массива и должно быть вычисляемым. См. «Вычисляемые операнды» ранее в этой главе. Направление **direction** должно быть **to** или **downto** . Направление скольжения должно быть таким же , как направление идентификатора массива. Если левое и правое выражение равны , определяется одиночный элемент. Значение , возвращаемое оператору , является подмассивом , содержащим определенные элементы массива. В примере 5-8 скользящие имена используются для назначения восьмибитового входа восьмибитовому выходу с обменом младших и старших четырех бит.

Пример 5-18. Скользящие имена в качестве операндов.

```
signal A, Z: BIT_VECTOR(0 to 7);

Z(0 to 3) <= A(4 to 7);
Z(4 to 7) <= A(0 to 3);
```



В примере 5-18 скользящие имена также используются как назначающие указатели. Такое применение описано в Главе 6 в разделе «Скользящие указатели».

Ограничения нулевого скольжения

FPGA Express не поддерживает нулевое скольжение , которое характеризуется нулевым диапазоном , таким как **(4 to 3)** , или диапазоном с неправильным направлением , таким как **UP_VAR (3 downto 2)** , если объявлен возрастающий диапазон **UP_VAR** (пример 5-19). В примере 5-19 показано три нулевых и одно не вычисляемое скольжение.

Пример 5-19. Нулевые и не вычисляемые скольжения.

```
subtype DOWN is         BIT_VECTOR(4 downto 0);
subtype UP is           BIT_VECTOR(0 to 7);
. . .
variable UP_VAR:        UP;
variable DOWN_VAR: DOWN;
. . .
UP_VAR(4 to 3)          -- Нулевое скольжение (нулевой диапазон)

UP_VAR(4 downto 0)   -- Нулевое скольжение (неправильное направление)
DOWN_VAR(0 to 1)     -- Нулевое скольжение (неправильное направление)
. . .

variable I: INTEGER range 0 to 7;
. . .
UP_VAR(I to I+1)        -- Не вычисляемое скольжение
```

Ограничения не вычисляемого скольжения

IEEE VHDL не разрешает не вычисляемые скольжения - скольжения , диапазон которых содержит не вычисляемое выражение.

Записи и поля

Записи состоят из именованных полей любого типа. Более подробную информацию см. в разделе «Типы запись» в Главе 4. В выражении вы можете сослаться на целую запись , либо на одиночное поле. Имена полей имеют следующий синтаксис :

record_name .field_name

где ***record_name*** - имя записи переменной или сигнала , а ***field_name*** - имя поля в записи этого типа. **field_name** отделяется от имени записи точкой (**.**). Заметим , что ***record_name*** отличается для каждой переменной или сигнала данного типа записи. ***field_name*** - это имя поля , определенное для данного типа записи. В примере 5-20 показано определение типа записи , а также обращение к записи и полю.

Пример 5-20. Обращение к записи и полю.

```
type BYTE_AND_IX is
        record
                BYTE: BIT_VECTOR(7 downto 0);
                IX:     INTEGER range 0 to 7;
        end record;

signal X: BYTE_AND_IX;
. . .
X               -- запись
X.BYTE          -- поле : 8-битовый массив
X.IX            -- поле : целое
```

Поле может быть любого типа , включая массив , запись или множество. Обращайтесь к элементу поля с нотацией этого типа , например :

```
X.BYTE(2)               -- один элемент из поля массива
BYTE
X.BYTE(3 downto 0)      -- 4-элементное скольжение по полю массива
BYTE
```

Множества

Множества могут рассматриваться как массивы литералов , поскольку они определяются типом массив и значением каждого элемента массива. Синтаксис множеств следующий :

type_name **' (** ***[choice =>] expression***
, [choice =>] expression })

Заметим , что синтаксис является более ограниченным , чем в Справочном руководстве по библиотеке (LRM). ***type_name*** - должно быть обязательно типа массив. Необязательный параметр ***choice*** определяет индекс элемента , последовательность индексов или другое. Каждое выражение ***expression*** обеспечивает значение для выбранного элемента и должно быть соответствующего типа. В примере 5-21 показано определение типа массив и множество , представляющее литерал этого типа. Два набора присвоений приводят к одному и тому же результату.

Пример 5-21. Простое множество.

```
subtype MY_VECTOR is BIT_VECTOR(1 to 4);
signal X:        MY_VECTOR;
variable A, B:   BIT;

X <= MY_VECTOR'('1', A nand B, '1', A or B)          – Присвоение множества
...
X(1) <= '1';                -- Присвоение по элементам
X(2) <= A nand B;
X(3) <= '1';
X(4) <= A or B;
```

Вы можете определить индекс элемента с помощью позиционной или именованной записи. При позиционной записи каждому элементу присваивается значение его порядкового выражения , как показано в примере 5-21. При использовании именованной записи конструкция **choice =>** определяет один или более элементов массива. Параметр **choice** может содержать выражение (такое , как **(Imod 2) =>**) для индикации индекса одиночного элемента или диапазон (такой , как **3 to 5 =>** или **7 downto 0 =>**) для индикации последовательности индексов.

Множества могут использовать как позиционную , так и именованную записи , однако позиционные выражения должны появляться перед именованными (***choice***). Нет необходимости определять все индексы элементов множества. Всем не назначенным элементам присваивается значение путем включения конструкции **others=>** ***expression*** как последнего элемента списка. В примере 5-22 показаны различные множества , представляющие одно и то же значение.

Пример 5-22. Эквивалентные множества.

```
subtype MY_VECTOR is BIT_VECTOR(1 to 4);

MY_VECTOR'('1', '1', '0', '0');
MY_VECTOR'(2 => '1', 3 => '0', 1 => '1', 4 => '0');
MY_VECTOR'('1', '1', others => '0');
MY_VECTOR'(3 => '0', 4 => '0', others => '1');
MY_VECTOR'(3 to 4 => '0', 2 downto 1 => '1');
```

Выражение **others** должно быть единственным выражением во множестве. В примере 5-23 показаны два эквивалентных множества.

Пример 5-23. Эквивалентные множества , использующие выражение **others** .

```
MY_VECTOR'(others => '1');
MY_VECTOR'('1', '1', '1', '1');
```

Для использования множества как указателя в операторе присваивания см. раздел «Множества - указатели» в Главе 6.

Атрибуты

VHDL определяет атрибуты для различных типов данных. Атрибут VHDL принимает переменную или сигнал данного типа и возвращает значение. Синтаксис атрибута следующий :

object'attribute

FPGA Express поддерживает следующие предопределенные атрибуты VHDL для использования с массивами , как описано в разделе «Типы массив» Главы 4 :

- **left**
- **right**
- **high**
- **low**
- **length**
- **range**
- **reverse_range**

FPGA Express также поддерживает следующие предопределенные атрибуты VHDL для использования с операторами **wait** и **if** , как описано в Главе 8 «Реализация регистров и третьего состояния» :

- **event**
- **stable**

В дополнение к поддерживаемым предопределенным атрибутам VHDL , перечисленным выше , FPGA Express определяет набор атрибутов , относящихся к синтезу. Эти специфические атрибуты FPGA Express могут быть расположены в вашем описании VHDL для прямой оптимизации. См. «Атрибуты и ограничения синтеза» в Главе 9 для более подробной информации.

Вызовы функций

Вызов функции выполняет именованную функцию с данными значениями параметров. Значение , возвращаемое оператору , является возвращаемым значением функции. Вызов функции имеет следующий синтаксис :

function_name **(** **[*parameter_name* =>]** ***expression***
{ , [*parameter_name* =>] ***expression*** **})**

где ***function_name*** - имя определенной функции. Необязательный параметр ***parameter_name*** является выражением формальных параметров , как определено в функции. Каждое выражение ***expression*** обеспечивает значение для своего параметра и должно иметь тип , соответствующий данному параметру.

Вы можете определять параметры с помощью позиционной или именованной записи так же , как значения множества. При позиционной записи конструкция ***parameter_name*** => опускается. Первое выражение обеспечивает значение для первого параметра функции , второе выражение - значение для второго параметра и т.д. При именованной записи перед выражением определяется конструкция ***parameter_name*** => ; именованный параметр получает значение этого выражения. Вы можете совместно использовать позиционные и именованные выражения в одном и том же вызове функции , до тех пор пока позиционные выражения стоят перед выражениями именованных параметров.

Вызовы функций осуществляются с помощью логики , пока вы не используете директиву компиляции **map_to_entity** . Для более подробной информации см. «Размещение подпрограмм в компоненты» в Главе 6 и «Директивы импликации компонентов» в Главе 9. В примере 5-24 показано объявление функции и несколько ее эквивалентных вызовов.

Пример 5-24. Вызовы функции.

```
function FUNC(A, B, C: INTEGER) return BIT;
. . .
FUNC(1, 2, 3)
FUNC(B => 2, A => 1, C => 7 mod 4)
FUNC(1, 2, C => -3+6)
```

Определенные выражения

Определенные выражения устанавливают тип операнда для разрешения неоднозначностей в определении типа операнда. Вы не можете использовать определенные выражения для *преобразования типа* (см. «Преобразования типа»). Определенные выражения имеют следующий синтаксис :

type_name**'(*****expression*****)**

где ***type_name*** - имя определенного типа. Выражение ***expression*** должно вычислять значение соответствующего типа.

Примечание : Одиночная кавычка , или апостроф , должна ставиться между type_name и (expression). Если кавычка опущена , то конструкция интерпретируется как преобразование типа.

В пример 5-25 показано определенное выражение , которое решает проблему перегрузки функции путем указания типа параметра как десятичного литерала.

Пример 5-25. Определенный десятичный литерал.

```
type R_1 is range 0 to 10;        -- Целое от 0 до 10
type R_2 is range 0 to 20;        -- Целое от 0 до 20

function FUNC(A: R_1) return BIT;
```

```
function FUNC(A: R_2) return BIT;

FUNC(5)              -- Неоднозначно ; должно быть типа R_1 , R_2 или INTEGER
FUNC(R_1'(5))        -- Однозначно
```

В примере 5-26 показано , как определенные выражения разрешают неоднозначности во множествах и перечисляемых литералах.

Пример 5-26. Определенные множества и перечисляемые литералы.

```
type ARR_1 is array(0 to 10) of BIT;
type ARR_2 is array(0 to 20) of BIT;
. . .
(others => '0')          -- Неоднозначно ; должно быть
                         -- типа ARR_1 или ARR_2
ARR_1'(others => '0')    -- Определено ; однозначно

type ENUM_1 is (A, B);
type ENUM_2 is (B, C);
. . .
B                        -- Неоднозначно ; должно быть
                         -- типа ENUM_1 или ENUM_2
ENUM_1'(B)               -- Определено ; однозначно
```

Преобразования типов

Преобразования типов изменяют тип выражения. Эти преобразования отличаются от определенных выражений , так как они изменяют тип своего выражения ; в то время как определенные выражения просто разрешают тип выражения. Преобразование типа имеет следующий синтаксис :

***type_name*(*expression*)**

где ***type_name*** - имя определенного типа. Выражение ***expression*** должно вычислять значение типа , который может быть преобразован в тип ***type_name.***

- Преобразования типов могут осуществляться между целыми типами и аналогичными типами массивов.
- Два типа массивов аналогичны , если они имеют одинаковую длину и преобразуемые или идентичные типы элементов.
- Перечисляемые типы не могут быть преобразованы.

В примере 5-27 показаны некоторые определения типов и соответствующие объявления сигналов , а также правильные и неправильные преобразования типов.

Пример 5-27. Правильные и неправильные преобразования типов.

```
type INT_1 is range 0 to 10;
type INT_2 is range 0 to 20;

type ARRAY_1 is array(1 to 10) of INT_1;
type ARRAY_2 is array(11 to 20) of INT_2;

subtype MY_BIT_VECTOR is BIT_VECTOR(1 to 10);
type BIT_ARRAY_10 is array(11 to 20) of BIT;
type BIT_ARRAY_20 is array(0 to 20) of BIT;

signal S_INT:        INT_1;
signal S_ARRAY:      ARRAY_1;
signal S_BIT_VEC:    MY_BIT_VECTOR;
```

```
signal S_BIT:          BIT;

                       -- Разрешенные преобразования типов
INT_2(S_INT)
                  -- Преобразование целого типа
BIT_ARRAY_10(S_BIT_VEC)
                  -- Преобразование массивов аналогичных типов

                       -- Не разрешенные преобразования типов
BOOLEAN(S_BIT);
                  -- Нельзя преобразовать перечисляемые типы друг в друга
INT_1(S_BIT);
                  -- Нельзя преобразовать перечисляемые типы в другие типы
BIT_ARRAY_20(S_BIT_VEC);
                  -- Длины массивов не равны
ARRAY_1(S_BIT_VEC);
                  -- Типы элементов не могут быть преобразованы
```

Глава 6. Последовательные операторы

Последовательные операторы , такие как **A := 3** воспринимаются один за другим в том порядке , в котором они записаны. Последовательные операторы VHDL могут появляться только в процессах или подпрограммах. Процесс VHDL является группой последовательных операторов ; подпрограмма является процедурой или функцией.

Для ознакомления с последовательными операторами обсудим следующее :

- Операторы присваивания
- Оператор присваивания переменной
- Оператор присваивания сигнала
- Оператор *if*
- Оператор *case*
- Операторы *loop*
- Оператор *next*
- Оператор *exit*
- Подпрограммы
- Оператор *return*
- Оператор *wait*
- Оператор *null*

Процессы состоят из последовательных операторов , однако сами процессы являются параллельными операторами (см. Главу 7). Все процессы в проекте выполняются параллельно. Однако , в любой данный момент времени внутри каждого процесса воспринимается только один последовательный оператор. Процесс связывается с остальной частью проекта с помощью читаемых или записываемых значений сигналов или портов , объявленных за пределами процесса. Последовательные алгоритмы могут быть выражены в виде подпрограмм и вызваны последовательно (как описано в этой главе) или параллельно (как описано в Главе 7).

Последовательными операторами являются :

операторы присваивания ,
: которые присваивают значения переменным и сигналам.

операторы управления программой ,
: которые условно выполняют операторы (***if*** и ***case***) , повторяют операторы (***for...loop***) и пропускают операторы (***next*** and ***exit***).

подпрограммы ,
: которые определяют последовательные алгоритмы для повторного использования в проекте (***procedure*** и ***function***).

оператор ожидания
: для паузы , пока не случится событие (***wait***).

нулевой оператор

для указания , что не нужны никакие действия (***null***).

Операторы присваивания

Операторы присваивания назначают значение переменной или сигналу и имеют следующий синтаксис :

***target* := *expression*;** **-- Назначение переменной**
***target* <= *expression*;** **-- Назначение сигналу**

где **target** (указатель) - переменная или сигнал (или часть переменной или сигнала , такая как подмассив) , который принимает значение выражения **expression** . Выражение должно вычислять значение того же типа , что и указатель. См. Главу 5 для более подробной информации о выражениях.

Разница в синтаксисе присваивания переменной и сигнала заключается в том , что для переменных используется знак **:=** , а для сигналов **<=**. Семантическое отличие в том , что переменные являются локальными величинами для процесса или подпрограммы , и их назначения имеют немедленный эффект. Сигналу могут не быть локальными для процесса или подпрограммы , и их назначения имеют эффект в конце процесса. Единственное , что означают сигналы , - это связь между процессами. Для более подробной информации о семантических различиях см. «Назначение сигналов» ниже в этой главе.

Назначаемые указатели

Операторы присваивания имеют пять типов указателей :

- Простые имена , как **my_var**
- Индексные имена , как **my_array_var(3)**
- Скользящие имена , как **my_array_var(3 to 6)**
- Именованные поля , как **my_record.a_field**
- Множества , как **(my_var1, my_var2)**

Назначаемый указатель может быть как переменной , так и сигналом ; ниже приведены описания для обоих.

Простые имена-указатели

Простые имена-указатели имеют следующий синтаксис присваивания :

***identifier* := *expression*;** **-- Назначение переменной**
***identifier* <= *expression*;** **-- Назначение сигналу**

где ***identifier*** - имя сигнала или переменной. Назначаемое выражение должно иметь тот же тип , что сигнал или переменная. Для типов массив значения присваиваются всем элементам массива. В примере 6-1 показаны некоторые назначения простым именам-указателям.

Пример 6-1. Простые имена-указатели.

```
variable A, B: BIT;
signal C: BIT_VECTOR(1 to 4);

-- Указатель    Выражение
     A :=           '1';        -- Переменной A присваивается '1'
     B :=           '0';        -- Переменной B присваивается '0'
     C <=           -1100";     -- Сигналу-массиву C присваивается -1100"
```

Индексные имена-указатели

Индексные имена-указатели имеют следующий синтаксис присваивания :

***identifier*(*index_expression*) := *expression*;**
-- Назначение переменной
***identifier*(*index_expression*) <= *expression*;**
-- Назначение сигналу

где ***identifier*** - имя сигнала или переменной типа массив. Выражение ***index_expression*** должно вычислять значение индекса для ***identifier*** соответствующего типа и в соответствующих границах , но не должно быть вычисляемым (см. «Вычисляемые операнды» в Главе 5) ; хотя в этом случае синтезируется дополнительное аппаратное обеспечение. Присваиваемое выражение ***expression*** должно содержать элемент типа массив.

В пример 6-2 элементам переменной-массива **A** назначаются значения в виде индексных имен.

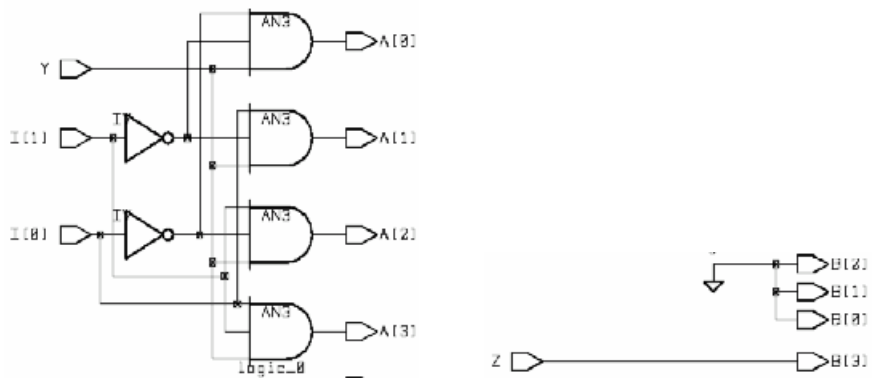
Пример 6-2. Индексные имена-указатели.

```
variable A: BIT_VECTOR(1 to 4);

-- Указатель    Выражение
   A(1) :=          '1';          -- Присваивает '1' первому элементу массива A.
   A(2) :=          '1';          -- Присваивает '1' второму элементу массива A.
   A(3) :=          '0';          -- Присваивает '0' третьему элементу массива A.
   A(4) :=          '0';          -- Присваивает '0' четвертому элементу массива A.
```

В примере 6-3 показаны два индексных имени-указателя. Один из указателей является вычисляемым , а второй - нет. Отметим различия в генерируемых аппаратных схемах для каждого из назначений.

Пример 6-3. Вычисляемые и не вычисляемые индексные имена-указатели.

```
signal A, B: BIT_VECTOR(0 to 3);
signal I: INTEGER range 0 to 3;
signal Y, Z: BIT;

A     <= -0000";
B     <= -0000";
A(I) <= Y;        -- Не вычисляемое индексное выражение
B(3) <= Z;        -- Вычисляемое индексное выражение
```



Скользящие указатели

Скользящие указатели имеют следующий синтаксис :

***identifier*(*index_expr_1 direction index_expr_2*)**

где ***identifier*** - имя сигнала или переменной типа массив. Каждое выражение ***index_expr*** должно значение индекса для ***identifier*** соответствующего типа и в соответствующих границах. Оба ***index_expr*** должны быть вычисляемыми (см. «Вычисляемые операнды» в Главе 5) и должны лежать внутри границ массива. Направление ***direction*** должно согласовываться с направлением индекса массива ***identifier*** , то есть — **to** или **downto** . Назначаемое выражение должно содержать элемент типа массив.

В примере 6-4 переменным-массивам **A** и **B** назначаются одинаковые значения.

Пример 6-4. Скользящие указатели.

```
variable A, B: BIT_VECTOR(1 to 4);

-- Указатель    Выражение
   A(1 to 2) :=      -11";      -- Присваивает -11" первым двум элементам массива A
   A(3 to 4) :=      -00";      -- Присваивает -00" последним двум элементам массива A
   B(1 to 4) :=      -1100";    -- Присваивает -1100" массиву B
```

Поля-указатели

Поля- указатели имеют следующий синтаксис :

identifier.field_name

где ***identifier*** - имя сигнала или переменной типа запись , а ***field_name*** - имя поля из записи этого типа , которому предшествует точка (**.**). Назначаемое выражение должно иметь тип идентифицируемого поля. Поле может быть любого типа , включая массив . запись или множество. В примере 6-5 показано присваивание значений полям записи переменных **A** и **B**.

Пример 6-5. Поля-указатели.

```
type REC is
        record
                NUM_FIELD: INTEGER range -16 to 15;
                ARRAY_FIELD: BIT_VECTOR(3 to 0);
        end record;

variable A, B: REC;

-- Указатель            Выражение
A.NUM_FIELD :=          -12;        -- Присваивает -12 записи A полю NUM_FIELD
```

```
A.ARRAY_FIELD :=       -0011";      -- Присваивает -0011" записи A полю ARRAY_FIELD
A.ARRAY_FIELD(3) :=     '1';        -- Присваивает '1' старшему значащему биту записи
                                    -- A поля ARRAY_FIELD
      B :=                A;        -- Присваивает значения записи A соответствующим
                                    -- полям B
```

Более подробную информацию о полях-указателях см. в разделе «Типы запись» Главы 4.

Множества-указатели

Множества-указатели имеют следующий синтаксис присваивания :

```
( [choice =>] identifier
 , [choice =>] identifier } ) := array_expression;
-- Назначение переменной

( [choice =>] identifier
 , [choice =>] identifier } ) <= array_expression;
-- Назначение сигналу
```

Множественное назначение присваивает значения элементов ***array_expression*** одной или более переменной или сигналу (***identifier***). Каждый необязательный параметр ***choice*** является индексным выражением , выбирающим элемент или диапазон скольжения при назначении ***array_expression***. Каждый идентификатор должен иметь тип элемента ***array_expression***. ***identifier*** может быть типа массив. В примере 6-6 приведены некоторые множества-указатели.

Пример 6-6. Множества-указатели.

```
signal A, B, C, D: BIT;
signal S: BIT_VECTOR(1 to 4);
. . .
variable E, F: BIT;
variable G: BIT_VECTOR(1 to 2);
variable H: BIT_VECTOR(1 to 4);

-- Позиционная запись
S               <= ('0', '1', '0', '0');
(A, B, C, D)    <= S;    -- Присваивает '0' в A , '1' в B , '0' в C и '0' в D

-- Именованная запись
(3 => E , 4 => F , 2 => G(1) , 1 => G(2)) := H;     -- Присваивает H(1) в G(2) , H(2) в G(1) ,
                                                    -- H(3) в E и H(4) в F
```

Вы можете присвоить значения элементов массива идентификаторам с помощью позиционной или именованной записи. При позиционной записи конструкция ***choice*** => не используется. Значения элементов массива назначаются идентификаторам , начиная от левой границы массива и заканчивая правой. При именованной записи конструкция ***choice*** => определяет отдельные элементы назначаемого массива. Индексное выражение ***choice*** индицирует одиночный элемент , например **3**. Тип идентификатора (***identifier***) должен согласовываться с типом назначаемого элемента.

Позиционная и именованная записи могут быть смешанными , однако позиционные элементы при этом должны появляться перед именованными.

Оператор присваивания переменной

Присваивание переменной изменяет значение переменной. Такое присваивание имеет следующий синтаксис :

***target* := *expression*;**

где ***expression*** определяет присваиваемое значение ; его тип должен быть совместим с типом указателя (***target***). См. Главу 5 для более подробной информации о выражениях. Указатель ***target*** именует переменную , которая принимает значение ***expression***. См. «Назначаемые указатели» в предыдущем разделе об описании назначаемых указателей переменной.

Когда переменной присваивается значение , то такое присвоение имеет немедленный эффект. Переменная хранит назначенное значение до тех пор , пока ей не будет присвоено новое.

Оператор присваивания сигнала

При назначении сигналу изменяется значение , управляемое сигналом в текущем процессе. Синтаксис присваивания сигналу следующий :

***target* <= *expression*;**

где ***expression*** определяет присваиваемое значение ; его тип должен быть совместимым с указателем ***target***. См. Главу 5 для более подробной информации о выражениях. Указатель ***target*** именует сигналы , которые принимают значение ***expression***. См. «Назначаемые указатели» в этой главе об описании назначаемых указателей сигнала.

Сигналы и переменные ведут себя по-разному , когда им присваиваются значения. Отличие заключается в том , какой эффект оказывают эти два типа назначений и как они влияют на чтение значений из переменных или сигналов.

Присваивание переменной

Когда переменной назначается какое-либо значение , то это имеет немедленный эффект. Переменная хранит это значение до тех пор , пока ей не будет присвоено новое.

Присваивание сигнала

Когда значение присваивается сигналу , то это необязательно имеет немедленный эффект , поскольку значение сигнала определяется процессами (или другими параллельными операторами) , которые ими управляют.

- Если несколько значений присваиваются данному сигналу в одном процессе , то только последнее назначение будет эффективным. Даже если сигнал в процессе назначен , прочитан и переназначен, читаемое значение (независимо , внутри или снаружи процесса) будет являться последним из назначенных.
- Если различные процессы (или другие параллельные операторы) присваивают значения одному сигналу , то источники будут соединены вместе. Результирующая схема зависит от конкретных выражений и целевой технологии. Она может быть недопустимой , объединяющим И , объединяющим ИЛИ либо шиной с третьим состоянием. См. «Управляемые сигналы» в Главе 7 для более подробной информации.

В примере 6-7 показаны различные эффекты при назначении переменной и сигнала.

Пример 6-7. Присваивания переменной и сигнала.

```
signal S1, S2: BIT;
signal S_OUT: BIT_VECTOR(1 to 8);
. . .
process( S1, S2 )
```

```
        variable V1, V2: BIT;
begin
        V1 := '1';              -- Устанавливается значение V1
        V2 := '1';              -- Устанавливается значение V2
        S1 <= '1';              -- Назначается источник для S1
        S2 <= '1';              -- Не имеет никакого эффекта , поскольку
                                -- переназначается позже в этом процессе
        S_OUT(1) <= V1;         -- Присваивает '1', ранее назначенное значение
        S_OUT(2) <= V2;         -- Присваивает '1', ранее назначенное значение
        S_OUT(3) <= S1;         -- Присваивает '1', ранее назначенное значение
        S_OUT(4) <= S2;         -- Присваивает '0', ранее назначенное значение

        V1 := '0';              -- Устанавливается новое значение V1
        V2 := '0';              -- Устанавливается новое значение V2
        S2 <= '0';              -- Это присваивание перезаписывает употребленное выше ,
                                -- так как это последнее назначение данного сигнала в данном
                                -- процессе
        S_OUT(5) <= V1;         -- Присваивает'0', ранее назначенное значение
        S_OUT(6) <= V2;         -- Присваивает'0', ранее назначенное значение
        S_OUT(7) <= S1;         -- Присваивает'1', ранее назначенное значение
        S_OUT(8) <= S2;         -- Присваивает'0', ранее назначенное значение
end process;
```

Оператор if

Оператор **if** выполняет последовательность операторов. Последовательность зависит от значения одного или более условий. Синтаксис оператора следующий :

```
if condition then
        { sequential_statement }
{ elsif condition then
        { sequential_statement } }
[ else
        { sequential_statement } ]
end if;
```

Каждое условие ***condition*** должно быть Булевским выражением. Каждое ветвление оператора **if** может иметь один или несколько последовательных операторов (***sequential_statement***).

Вычисляемое *условие*

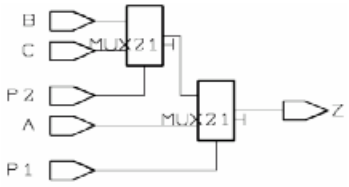
Оператор **if** вычисляет каждое условие (***condition***) по порядку. Первое (и только первое) истинное условие (**TRUE**) обуславливает выполнение операторов в ветвлении. Оставшаяся часть оператора **if** пропускается. Если ни одно из условий не является истинным , и присутствует предложение **else** , то выполняются включенные в него операторы. Если ни одно из условий не является истинным , и отсутствует предложение **else** , то не выполняется ни один оператор.

В пример 6-8 показан оператор **if** и соответствующая схема.

Пример 6-8. Оператор if .

```
signal A, B, C, P1, P2, Z: BIT;

if (P1 = '1') then
        Z <= A;
elsif (P2 = '0') then
        Z <= B;
else
```

```
        Z <= C;
end if;
```



Использование оператора *if* для реализации регистров и защелок

Некоторые формы оператора **if** могут быть использованы подобно оператору **wait** для тестирования срезов сигнала и , таким образом , реализации синхронной логики. Такое использование заставляет FPGA Express реализовывать регистры или защелки , как описано в Главе 8 «Реализация регистров и третьего состояния».

Оператор case

Оператор **case** выполняет одну из нескольких последовательностей операторов в зависимости от значения одиночного выражения. Синтаксис оператора следующий :

```
case expression is
        when choices =>
                { sequential_statement }
        { when choices =>
                { sequential_statement } }
end case;
```

где выражение **expression** должно вычислять значение целого (**INTEGER**) , или перечисляемого , или массива перечисляемых типов , таких как **BIT_VECTOR** . Каждый из параметров ***choices*** должен иметь вид

```
choice { | choice }
```

Каждый параметр **choice** может быть либо статическим выражением (как **3**) , либо статическим диапазоном (как **1 to 3**). Тип выражения **choice** определяет тип каждого параметра **choice.** Каждое значение из диапазона выражения **choice** должно покрываться одним выбором **choice.** Последний **choice** может быть **others** , при этом происходит согласование всех оставшихся (невыбранных) значений из диапазона типа выражения. Выбор **others** , если он присутствует , согласует выражение только если никакой другой из выборов не согласован с ним.

Оператор **case** вычисляет выражение **expression** и сравнивает его со значениями каждого выбора **choice**. Операторы , следующие за каждым предложением **when** , вычисляются только в том случае , если значение **choice** согласуется со значением **expression**.

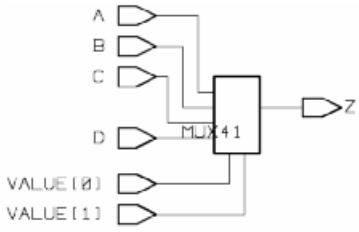
На выбор накладываются следующие ограничения :

- Никакие два выбора не могут перекрываться.
- Если нет выбора **others** , то все возможные значения выражения **expression** должны перекрываться соответствующим набором выборов.

В примере 6-9 показан оператор **case** , который выбирает один из четырех операторов назначения сигнала с помощью выражения перечисляемого типа.

Пример 6-9. Оператор case , который использует перечисляемый тип.

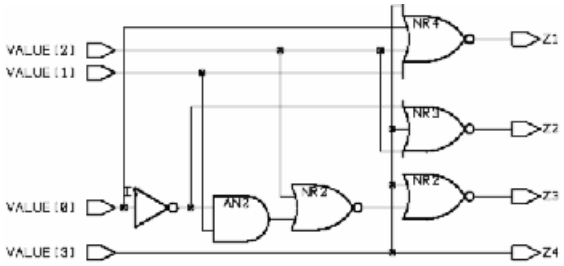
```
type ENUM is (PICK_A, PICK_B, PICK_C, PICK_D);
signal VALUE: ENUM;

signal A, B, C, D, Z: BIT;

case VALUE is
        when PICK_A =>
                Z <= A;
        when PICK_B =>
                Z <= B;
        when PICK_C =>
                Z <= C;
        when PICK_D =>
                Z <= D;
end case;
```



В примере 6-10 показан оператор **case** , вновь используемый для выбора одного из четырех операторов назначения сигнала , но на этот раз с помощью использования выражения целого типа с несколькими выборами.

Пример 6-10. Оператор case с целым типом.

```
signal VALUE is INTEGER range 0 to 15;
signal Z1, Z2, Z3, Z4: BIT;

Z1 <= '0';
Z2 <= '0';
Z3 <= '0';
Z4 <= '0';

case VALUE is
        when 0 =>               -- Согласует 0
                Z1 <= '1';
        when 1 | 3 =>           -- Согласует с 1 по 3
                Z2 <= '1';
        when 4 to 7 | 2 =>      -- Согласует 2, 4, 5, 6 или 7
                Z3 <= '1';
        when others =>          -- Согласует остальные значения с 8 по 15
                Z4 <= '1';
end case;
```



Недопустимые операторы *case*

В примере 6-11 показано четыре недопустимых оператора выбора.

Пример 6-11. Недопустимые операторы ***case***

```
signal VALUE: INTEGER range 0 to 15;
signal OUT_1: BIT;

case VALUE is            -- Должно быть по меньшей мере одно предложение
end case;

case VALUE is                        -- Значения с 2 по 15 не перекрыты выборами
        when 0 =>
                OUT_1 <= '1';
        when 1 =>
                OUT_1 <= '0';
end case;

case VALUE is                        -- Выборы с 5 по 10 перекрываются друг с другом
        when 0 to 10 =>
                OUT_1 <= '1';
        when 5 to 15 =>
                OUT_1 <= '0';
end case;
```

Операторы loop

Оператор **loop** повторно выполняет последовательность операторов. Синтаксис оператора следующий :

```
[label :] [iteration_scheme] loop
        { sequential_statement }
        { next [ label ] [ when condition ] ; }
        { exit [ label ] [ when condition ] ; }
end loop [label];
```

Необязательный параметр ***label*** именует цикл и пригоден для построения вложенных циклов. Каждый тип схемы итерации ***iteration_scheme*** описан в данном разделе. Операторы **next** и **exit** являются последовательными и используются только внутри циклов. Оператор **next** пропускает оставшуюся часть текущего цикла и продолжает работу , начиная со следующей итерации. Оператор **exit** пропускает оставшуюся часть текущего цикла и продолжает работу , начиная со следующего оператора , стоящего после цикла.

VHDL обеспечивает три типа операторов цикла , каждый из которых имеет свою схему итерации :

loop

Базовый оператор **loop** не имеет схемы итерации. Включенные в него операторы выполняются повторно до тех пор , пока не встретится оператор **exit** или **next**.

while .. loop

Оператор **while .. loop** имеет Булевскую схему итерации. Если вычисленное условие итерации имеет истинное значение , то включенные в цикл операторы выполняются один раз. Условие итерации затем вычисляется снова. Пока условие итерации остается истинным , цикл выполняется повторно. Когда вычисленное условие итерации становится ложным , цикл пропускается , и выполнение программы продолжается , начиная с оператора , стоящего вслед за циклом.

for .. loop

Оператор **for .. loop** имеет целую схему итерации , при которой количество повторов определяется целым диапазоном. Цикл повторяется один раз для каждого значения диапазона. После того , как будет достигнуто последнее значение диапазона итерации, цикл пропускается , и выполнение программы продолжается , начиная с оператора , стоящего вслед за циклом.

Предупреждение : Не вычисляемые циклы (операторы loop и while..loop) должны иметь как минимум один оператор wait в каждом включенном логическом ветвлении. В противном случае создается комбинационная петля обратной связи. См. «Оператор wait» позже в этой главе для более подробной информации. Обратно , вычисляемые циклы (операторы for..loop) не должны содержать операторы wait. В противном случае может возникнуть состояние состязания.

Оператор *loop*

Оператор **loop** без схемы итерации повторяет включенные в него операторы неопределенное количество раз. Синтаксис этого оператора следующий :

***[label* :]** **loop**
{ sequential_statement }
end loop *[label]*;

Необязательный параметр ***label*** именует этот цикл. Последовательные операторы ***sequential_statement*** могут быть любыми из описанных в этой главе. Два последовательных оператора используются только внутри циклов : оператор **next , который пропускает оставшуюся часть текущей итерации цикла , и exit** , который завершает цикл. Эти операторы описаны в следующих двух разделах.

Примечание : Оператор loop должен иметь по меньшей мере один оператор wait на каждое включенное логическое ветвление. См. «Оператор wait» позже в этой главе.

Оператор while .. loop

Оператор **while..loop** повторяет включенные в него операторы до тех пор , пока его условие итерации является истинным. Синтаксис этого оператора следующий :

***[label* :]** **while *condition* loop**
{ sequential_statement }
end loop *[label]*;

Необязательный параметр ***label*** именует этот цикл. Условие ***condition*** является любым Булевским выражением , таким как **((A = '1') or (X < Y))** . Последовательные операторы ***sequential_statement*** могут быть любыми из описанных в этой главе. Два последовательных оператора используются только внутри циклов : оператор **next , который пропускает оставшуюся часть текущей итерации цикла , и exit** , который завершает цикл. Эти операторы описаны в следующих двух разделах.

Примечание : Оператор while..loop должен иметь по меньшей мере один оператор wait на каждое включенное логическое ветвление. См. «Оператор wait» позже в этой главе.

Оператор for .. loop

Оператор **for..loop** повторяет включенные в него операторы один раз для каждого значения в целом диапазоне. Синтаксис этого оператора следующий :

[label* :]** **for** ***identifier **in** ***range*** **loop**
{ sequential_statement }
end loop ***[label]*;**

Необязательный параметр ***label*** именует этот цикл. Использование идентификатора ***identifier*** является специфическим для оператора **for..loop** :

- ***identifier*** не объявляется где-либо еще. Он объявляется автоматически самим циклом и является для него локальным. Идентификатор цикла перекрывает любой другой идентификатор с таким же именем , но только внутри цикла.
- Значение идентификатора может быть прочитано только внутри цикла (за пределами цикла он не существует). Вы не можете присвоить значение идентификатору цикла.

FPGA Express требует , чтобы диапазон ***range*** был вычисляемым целым выражением (см. «Вычисляемые операнды» в Главе 5) в одном из двух видов :

integer_expression **to** ***integer_expression***
integer_expression **downto** ***integer_expression***

Последовательные операторы ***sequential_statement*** могут быть любыми из описанных в этой главе. Два последовательных оператора используются только внутри циклов : оператор **next , который пропускает оставшуюся часть текущей итерации цикла , и exit** , который завершает цикл. Эти операторы описаны в следующих двух разделах.

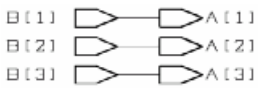
Примечание : Оператор for..loop не должен содержать какие-либо операторы wait.

Оператор **for .. loop** выполняется следующим образом :
1. Новая , локальная , целая переменная объявляется с именем ***identifier***.
2. Идентификатору назначается первое значение диапазона ***range*** , и последовательность операторов выполняется один раз.
3. Идентификатору назначается следующее значение диапазона , и последовательность операторов выполняется еще раз.
4. Шаг 3 повторяется , пока идентификатору не будет назначено последнее значение диапазона. Последовательность операторов затем выполняется в последний раз , и работа программы продолжается , начиная с оператора , стоящего за **end loop** . После этого цикл становится недоступным.

В примере 6-12 показано два эквивалентных фрагмента программы.

Пример 6-12. Оператор for..loop в эквивалентных фрагментах.

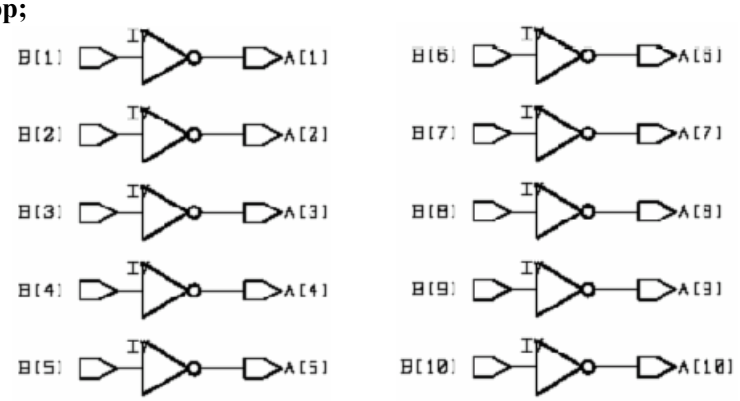
```
variable A, B: BIT_VECTOR(1 to 3);

-- Первый фрагмент является оператором цикла
for I in 1 to 3 loop
        A(I) <= B(I);
end loop;

-- Во втором фрагменте определяются три эквивалентных оператора
A(1) <= B(1);
A(2) <= B(2);
A(3) <= B(3);
```



Вы можете использовать оператор **loop** для работы со всеми элементами массива независимо от его размера. В примере 6-13 показано , как может быть использован атрибут массива **'range** — в данном случае для инвертирования каждого элемента битового вектора **A**.

Пример 6-13. Оператор for..loop работает с целым массивом.

```
variable A, B: BIT_VECTOR(1 to 10);
. . .
for I in A'range loop
        A(I) := not B(I);
end loop;
```



Неограниченные массивы и их атрибуты описаны в разделе «Типы массив» Главы 4.

Оператор next

Оператор **next** завершает текущую итерацию цикла , а затем продолжает его , начиная с первого оператора. Синтаксис этого оператора следующий :

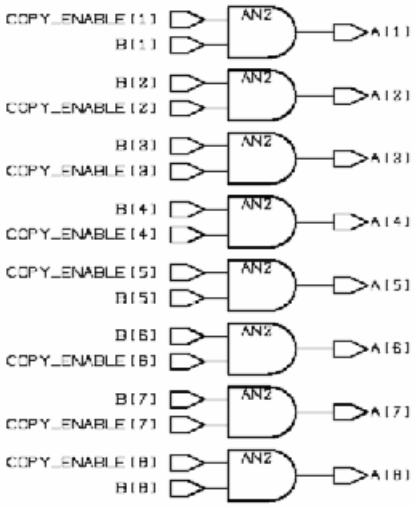
next ***[label] [*** **when** ***condition]*** **;**

Оператор **next** без метки ***label*** завершает текущую итерацию самого глубокого из вложенных циклов. Если вы определяете метку ***label*** , то завершается текущая итерация этого именованного цикла. Необязательное предложение **when** выполняет оператор **next** при определенном условии ***condition*** (Булевское выражение) , которое является истинным.

В примере 6-14 оператор **next** используется для условного копирования бит из вектора **B** в вектор **A** .

Пример 6-14. Оператор next.

```
signal A, B, COPY_ENABLE: BIT_VECTOR (1 to 8);
. . .
A <= -00000000";
. . .
-- B присваивается значение , такое как -01011011"
-- COPY_ENABLE присваивается значение , такое как -11010011"
. . .
for I in 1 to 8 loop
        next when COPY_ENABLE(I) = '0';
        A(I) <= B(I);
end loop;
```



В примере 6-15 показано использование вложенных операторов **next** в именованных циклах. В этом примере обрабатываются :

- Первый элемент вектора **X** против первого элемента вектора **Y**,
- Второй элемент вектора **X** против каждого из первых двух элементов вектора **Y**,
- Третий элемент вектора **X** против каждого из первых трех элементов вектора **Y**.

Пример 6-15. Именованный оператор next .

```
signal X, Y: BIT_VECTOR(0 to 7);
A_LOOP: for I in X'range loop
. . .
B_LOOP: for J in Y'range loop
. . .
next A_LOOP when I < J;
. . .
end loop B_LOOP;
. . .
end loop A_LOOP;
```

Оператор exit

Оператор **exit** завершает цикл. Выполнение программы продолжается , начиная с оператора , стоящего за **end loop** . Синтаксис этого оператора следующий :

exit ***[label] [*** **when** ***condition]*** **;**

Оператор **exit** без метки ***label*** завершает самый глубокий из вложенных циклов. Когда вы определяете метку ***label*** , то завершается данный именованный цикл , как показано раньше в примере 6-15. Необязательное предложение **when** выполняет свой оператор **exit** , если условие ***condition*** (Булевское выражение) является истинным.

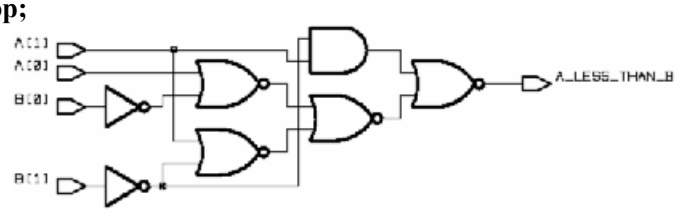
Операторы **exit** и **next** являются эквивалентными конструкциями. Оба оператора используют идентичный синтаксис и оба пропускают оставшуюся часть включенного (или именованного) цикла. Единственное отличие между ними заключается в том , что **exit** завершает свой цикл, а **next** продолжает его со следующей итерации (если таковая существует).

В примере 6-16 сравниваются два битовых вектора. Оператор **exit** совершает выход из цикла сравнения в тот момент , когда обнаруживается отличие.

Пример 6-16. Компаратор , использующий оператор exit .

```
signal A, B:                BIT_VECTOR(1 downto 0);
```

```
signal A_LESS_THAN_B:        Boolean ;
. . .
A_LESS_THAN_B <= FALSE;

for I in 1 downto 0 loop
        if (A(I) = '1' and B(I) = '0') then
                A_LESS_THAN_B <= FALSE;
                exit;
        elsif (A(I) = '0' and B(I) = '1') then
                A_LESS_THAN_B <= TRUE;
                exit;
        else
                null;     -- Сравнение продолжается
        end if;
end loop;
```



Подпрограммы

Подпрограммы являются независимыми именованными алгоритмами. Подпрограмма является процедурой **procedure** (ноль или более параметров **in** , **inout** или **out**) или функцией **function** (ноль или более параметров **in** и одно возвращаемое значение **return**).

Подпрограммы вызываются по имени в любом месте внутри архитектуры или тела блока объявлений VHDL. Подпрограммы могут быть вызваны последовательно (как описано позже в этой главе) или параллельно (как описано в Главе 7). В терминах аппаратного обеспечения вызов подпрограммы аналогичен реализации модуля , за исключением тех случаев , когда такой вызов становится частью текущей схемы , в то время как реализация модуля добавляет уровень иерархии в проект. Синтезированная подпрограмма всегда является комбинационной схемой (используйте **process** для создания последовательной схемы).

Подпрограммы , как и блоки объявлений , имеют свои объявления и тела. Объявление подпрограммы определяет ее имя , параметры и возвращаемое значение (для функций). Тело подпрограммы выполняет нужные вам операции. Часто блок объявлений содержит только тип и объявления подпрограммы для использования в других блоках. Тела объявленных подпрограмм затем реализуются в телах блоков объявлений. Преимущество разделения между блоками и телами заключается в том , что интерфейсы подпрограмм могут быть объявлены в публичных блоках объявлений в процессе развития системы. Одна группа разработчиков может использовать общие подпрограммы , в то время как другая группа будет создавать соответствующие тела. Вы можете модифицировать тела блоков объявлений , включая тела подпрограмм , без воздействия на других пользователей этих объявлений. Вы также можете определить подпрограммы локально , внутри объекта , блока или процесса.

FPGA Express выполняет вызовы процедур и функций с помощью комбинационной логики , если вы используете директиву компилятора **map_to_entity** (см. «Размещение подпрограмм в компоненты» позже в этой главе). FPGA Express не разрешает реализацию последовательных приборов , таких как защелки и триггера , в подпрограммах.

В примере 6-17 показан блок объявлений , содержащий тела и объявления некоторых процедур и функций. Сам по себе этот пример не является синтезируемым ; он просто создает шаблон. Тем не менее , проекты , которые реализуют процедуру **P** , компилируются нормально.

Пример 6-17. Объявления и тела подпрограмм.

```
package EXAMPLE is
        procedure P (A: in INTEGER; B: inout INTEGER);
```

```
        -- Объявление процедуры P

        function INVERT (A: BIT) return BIT;
        -- Объявление функции INVERT
end EXAMPLE;

package body EXAMPLE is
        procedure P (A: in INTEGER; B: inout INTEGER) is
        -- Тело процедуры P
        begin
                B := A + B;
        end;

        function INVERT (A: BIT) return BIT is
        -- Тело функции INVERT
        begin
                return (not A);
        end;
end EXAMPLE;
```

Для более подробной информации о подпрограммах см раздел «Подпрограммы» в Главе 3.

Вызовы подпрограмм

Подпрограммы могут иметь ноль или более параметров. Объявление подпрограммы определяет имя , режим и тип каждого параметра. Это формальные параметры подпрограммы. Когда подпрограмма вызывается , каждому формальному параметру дается значение , называемое *действительным параметром*. Значение каждого действительного параметра (соответствующего типа) может происходить из выражения , переменной или сигнала.

Режим параметра определяет , может ли действительный параметр быть прочитан (режим **in**) , записан (режим **out**) или прочитан и записан (режим **inout**). Действительные параметры , использующие режимы **out** и **inout** , должны быть переменными или сигналами , включая индексные (**A(1)**) и скользящие (**A(1 to 3)**) имена , но не могут являться константами или выражениями.

Процедуры и функции - это два типа подпрограмм :

процедура

Может иметь несколько параметров , которые используют режимы **in** , **inout** и **out**. Сама по себе не возвращает значение. Процедуры используются , если вы хотите обновить некоторые параметры (режимы **out** и **inout**) или если вам не нужно возвращать значение. Примером может служить процедура с одним битовым вектором **inout** , которая инвертирует каждый его бит.

функция

Может иметь несколько параметров , но только в режиме **in**. Возвращает собственное значение функции. Часть объявления функции определяет тип возвращаемого значения (также называемого *типом функции)*. Функции используются , если вам не нужно обновлять параметры и вы хотите возвратить одиночное значение. Например , арифметическая функция **ABS** возвращает абсолютное значение своего параметра.

Вызовы процедур

Вызов процедуры выполняет именованную процедуру с данными параметрами. Синтаксис этого вызова следующий :

procedure_name [([name =>] expression
{ , [name =>] expression })] ;

Каждое выражение ***expression*** называется действительным параметром ; ***expression*** часто является просто идентификатором. Если присутствует имя ***name*** (позиционная запись) , то имя формального параметра ассоциируется с выражением действительного параметра. Формальные параметры согласуются с действительными через именованную или позиционную запись. Эти записи могут быть смешанными , однако позиционные параметры должны появляться перед именованными.

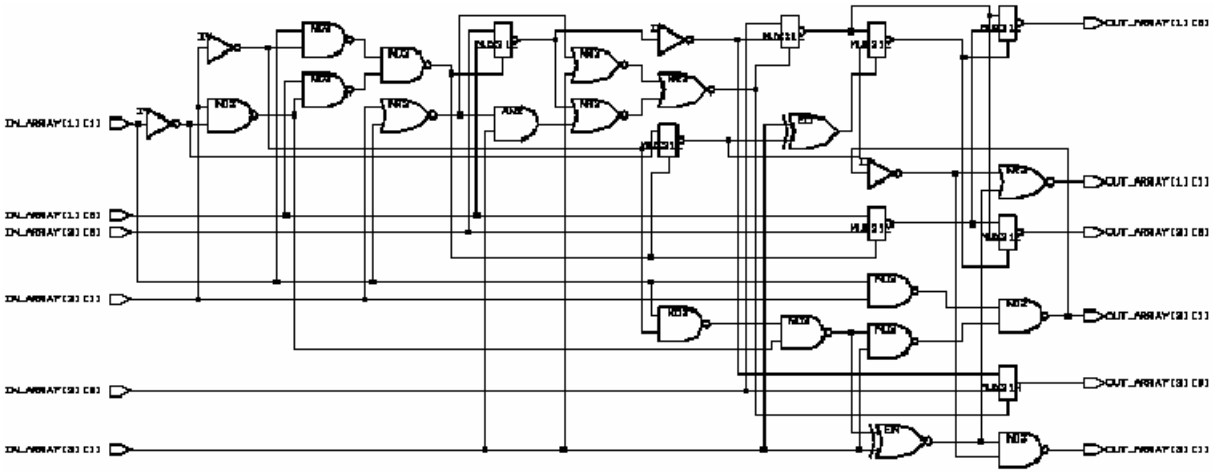
Концептуально вызов процедуры выполняется за три шага. Во-первых , значения действительных параметров **in** и **inout** присваиваются соответствующим формальным параметрам. Во-вторых , выполняется сама процедура. В-третьих , значения формальных параметров **inout** и **out** присваиваются действительным параметрам.

В синтезируемом аппаратном обеспечении действительные входы и выходы процедуры соединяются с ее внутренней логикой. В примере 6-18 показана локальная процедура с именем **SWAP** , которая сравнивает два элемента массива и меняет их местами , если они стоят не по порядку. **SWAP** вызывается повторно для сортировки массива из трех чисел.

Пример 6-18. Вызов процедуры для сортировки массива.

```
package DATA_TYPES is
        type DATA_ELEMENT is range 0 to 3;
        type DATA_ARRAY is array (1 to 3) of DATA_ELEMENT;
end DATA_TYPES;

use WORK.DATA_TYPES.ALL;
entity SORT is
        port(   IN_ARRAY:           in DATA_ARRAY;
                OUT_ARRAY:          out DATA_ARRAY);
end SORT;

architecture EXAMPLE of SORT is
begin

        process(IN_ARRAY)
                procedure SWAP(     DATA: inout DATA_ARRAY;
                                    LOW, HIGH:   in INTEGER)   is
                        variable TEMP: DATA_ELEMENT;
                begin
                        if(DATA(LOW) > DATA(HIGH)) then          -- Проверка данных
                                TEMP := DATA(LOW);
                                DATA(LOW) := DATA(HIGH);         -- Перестановка данных
                                DATA(HIGH) := TEMP;
                        end if;
                end SWAP;
        variable MY_ARRAY: DATA_ARRAY;

        begin
                MY_ARRAY := IN_ARRAY;   -- Чтение входного значения в переменную

                -- Парная сортировка
                SWAP(MY_ARRAY, 1, 2);       -- Перестановка первого и второго
                SWAP(MY_ARRAY, 2, 3);       -- Перестановка второго и третьего
                SWAP(MY_ARRAY, 1, 2);       -- Перестановка первого и второго
```

```
again
        OUT_ARRAY <= MY_ARRAY;          -- Запись результата на выход
        end process;
end EXAMPLE;
```



Вызовы функций

Вызов функции аналогичен вызову процедуры , за исключением того , что вызов функции является типом выражения , поскольку он возвращает значение. В примере 6-19 показано определение простой функции и два ее вызова.

Пример 6-19. Вызов функции.

```
function INVERT (A : BIT) return BIT is
        begin
                return (not A);
        end;
...
process
        variable V1, V2, V3: BIT;
begin
        V1 := '1';
        V2 := INVERT(V1) xor 1;
        V3 := INVERT('0');
end process;
```

Для более подробной информации см. «Вызовы функций» в разделе «Операнды» в Главе 5.

Оператор return

Оператор **return** завершает подпрограмму. Этот оператор обязателен при определении функций и необязателен при определении процедур. Синтаксис этого оператора следующий :

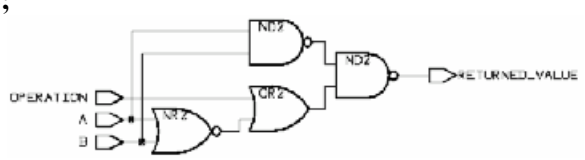
```
return expression ;     -- Функции
return ;                -- Процедуры
```

Требуемое выражение ***expression*** обеспечивает возвращаемое значение функции. Каждая функция должна иметь по меньшей мере один оператор **return** . Тип выражения должен согласовываться с объявленным типом функции. Функция может иметь более одного оператора **return** . Только один оператор **return** достигается при данном вызове функции. Процедура может иметь один или более операторов **return** , однако для нее не разрешено выражение ***expression*** . Оператор **return** , если таковой присутствует , является последним выполняемым оператором в процедуре.

В примере 6-20 функция **OPERATE** возвращает логическое И либо ИЛИ своих параметров **A** и **B**. Возврат зависит от значения параметра **OPERATION** .

Пример 6-20. Использование нескольких операторов return .

```
function OPERATE(A, B, OPERATION: BIT) return BIT is
begin
        if (OPERATION = '1') then
                return (A and B);
        else
                return (A or B);
        end if;
end OPERATE;
```



Размещение подпрограмм в компоненты (объекты)

В VHDL объекты не могут вызываться изнутри поведенческого кода. Процедуры и функции не могут существовать как объекты (компоненты) , а должны представляться логическими элементами. Вы можете преодолеть это ограничение с помощью директивы компилятора **map_to_entity** , которая заставляет FPGA Express выполнять функции или процедуры в виде компонентной реализации. Процедуры и функции , которые используют **map_to_entity** , представляются в виде компонентов в тех проектах , в которых они вызываются. Вы также можете использовать окно выполнения FPGA Express (Implementation Window) для создания нового уровня иерархии из подпрограммы VHDL , как описано в *Руководстве пользователя FPGA Express.*

Когда вы добавляете директиву **map_to_entity** к определению подпрограммы , FPGA Express принимает существование объекта с идентифицируемым именем и таким же интерфейсом. FPGA Express не проверяет такое предположение до тех пор , пока не начинается линкование с корневым проектом. Согласованный объект должен иметь такие же имена входных и выходных портов. Если подпрограмма является функцией , вы также должны обеспечить директиву **return_port_name** , где согласованный объект имеет выходной порт с таким же именем. Эти две директивы называются директивами импликации компонента :

```
-- pragma map_to_entity entity_name
-- pragma return_port_name port_name
```

Вставьте эти директивы после определения процедуры или функции. Например :

```
function MUX_FUNC(A,B: in TWO_BIT; C: in BIT)
        return
                TWO_BIT is
-- pragma map_to_entity MUX_ENTITY
-- pragma return_port_name Z
...
```

Когда FPGA Express встречает директиву **map_to_entity** , он анализирует , но игнорирует содержимое объявления подпрограммы. Используйте **--pragmatranslate_off** и **--pragmatranslate_on** для укрытия специфических для моделирования конструкций внутри подпрограммы **map_to_entity**.

Примечание : Согласованный объект (entity_name) можно не записывать на VHDL. Он может находиться в любом формате , который поддерживает FPGA Express.
Предупреждение : Поведенческое описание подпрограммы не сравнивается с функциональным поведением объекта , который ее перегружает. Результаты моделирования до и после синтеза могут не

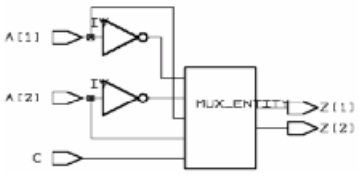
согласовываться , если существуют отличия в функционировании между подпрограммой VHDL и перегруженным объектом.

В примере 6-21 показана функция , которая использует директивы импликации компонента.

Пример 6-21. Использование директив импликации компонента в функции.

```
package MY_PACK is
        subtype TWO_BIT is BIT_VECTOR(1 to 2);
        function MUX_FUNC(A,B: in TWO_BIT; C: in BIT)
return
        TWO_BIT;
end;

package body MY_PACK is

        function MUX_FUNC(A,B: in TWO_BIT; C: in BIT)
return
        TWO_BIT is
-- pragma map_to_entity MUX_ENTITY
-- pragma return_port_name Z

-- содержимое этой функции игнорируется , но должно согласовываться с функциональным
-- поведением модуля MUX_ENTITY , тогда результаты моделирования будут согласованы
        begin
                if(C = '1') then
                        return(A);
                else
                        return(B);
                end if;
        end;
end;

use WORK.MY_PACK.ALL;

entity TEST is
        port(A: in TWO_BIT; C: in BIT; TEST_OUT: out TWO_BIT);
end;

architecture ARCH of TEST is
begin
        process
        begin
                TEST_OUT <= MUX_FUNC(not A, A, C);
                                -- Вызов импликации компонента
        end process;
end;
use WORK.MY_PACK.ALL;

-- следующий объект «перегружает» определенную выше функцию MUX_FUNC

entity MUX_ENTITY is
        port(A, B: in TWO_BIT; C: in BIT; Z: out TWO_BIT);
end;

architecture ARCH of MUX_ENTITY is
begin
        process
```

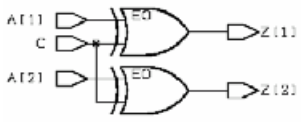
```
        begin
                case C is
                        when '1' => Z <= A;
                        when '0' => Z <= B;
                end case;
        end process;
end;
```



В примере 6-22 показан тот же проект , что и в примере 6-21 , но без создания объекта для функции. Директивы компилятора удалены.

Пример 6-22. Использование логических элементов для реализации функции.

```
package MY_PACK is
        subtype TWO_BIT is BIT_VECTOR(1 to 2);
        function MUX_FUNC(A,B: in TWO_BIT; C: in BIT)
                return TWO_BIT;
end;

package body MY_PACK is

        function MUX_FUNC(A,B: in TWO_BIT; C: in BIT)
                return TWO_BIT is
        begin
                if(C = '1') then
                        return(A);
                else
                        return(B);
                end if;
        end;
end;

use WORK.MY_PACK.ALL;
entity TEST is
        port(A: in TWO_BIT; C: in BIT; Z: out TWO_BIT);
end;

architecture ARCH of TEST is
begin
        process
        begin
                Z <= MUX_FUNC(not A, A, C);
        end process;
end;
```



Оператор wait

Оператор **wait** приостанавливает процесс до тех пор , пока не будет детектирован положительный или отрицательный срез сигнала. Синтаксис этого оператора следующий :

```
wait until signal = value ;
wait until signal'event and signal = value ;
wait until not signal'stable
        and signal = value ;
```

Здесь ***signal*** - имя однобитового сигнала — сигнала перечисляемого типа , кодируемого одним битом (см. «Кодирование перечисления» в Главе 4). Значение ***value*** должно быть одним из литералов перечисляемого типа. Если сигнал имеет тип **BIT** , то ожидаемое ***value*** является либо **'1'** для положительного среза , либо **'0'** для отрицательного среза.

Примечание : Три вида оператора wait из поднабора IEEE VHDL являются специфическими для текущей реализации FPGA Express.

Реализация синхронной логики

Оператор **wait** реализует синхронную логику , в которой ***signal*** обычно является тактирующим сигналом. В следующем разделе описано , как FPGA Express реализует эту логику.

В примере 6-23 показано три эквивалентных оператора **wait** (все тактируются положительным срезом).

Пример 6-23. Эквивалентные операторы wait .

```
wait until CLK = '1';
wait until CLK'event and CLK = '1';
wait until not CLK'stable and CLK = '1';
```

Когда синтезируется схема , аппаратное обеспечение для всех трех видов оператора **wait** будет одинаковым.

В примере 6-24 показан оператор **wait** , используемый для приостановки процесса , пока не будет детектирован следующий положительный срез (переход 0-в-1) сигнала **CLK**.

Пример 6-24. wait для положительного среза.

```
signal CLK: BIT;
...
process
begin
        wait until CLK'event and CLK = '1';
        -- Ожидание положительного перехода (среза)
        ...
end process;
```

Примечание : IEEE VHDL требует , чтобы у процесса , содержащего оператор wait , не было списка чувствительности. См. «Операторы процесса» в Главе 7 для более подробной информации.

В примере 6-25 показано , как оператор **wait** , используется для описания схемы , в которой значение инкрементируется по каждому положительному срезу синхроимпульса.

Пример 6-25. Цикл , использующий оператор wait .

```
process
begin
        y <= 0;
        wait until (clk'event and clk = '1');
```

```
        while (y < MAX) loop
                wait until (clk'event and clk = '1');
                x <= y ;
                y <= y + 1;
        end loop;
end process;
```

В примере 6-26 показано , как несколько операторов **wait** описывают многоцикловую схему , которая обеспечивает среднее значение входа **A** за четыре цикла синхроимпульсов.

Пример 6-26. Использование нескольких операторов wait .

```
process
begin
        wait until CLK'event and CLK = '1';
        AVE <= A;
        wait until CLK'event and CLK = '1';
        AVE <= AVE + A;
        wait until CLK'event and CLK = '1';
        AVE <= AVE + A;
        wait until CLK'event and CLK = '1';
        AVE <= (AVE + A)/4;
end process;
```

В примере 6-27 приведено два эквивалентных описания. Первое описание использует неявную логику состояния , а второй - явную логику состояния.

Пример 6-27. Операторы ***wait*** и логика состояния.

```
-- Неявная логика состояния
process
begin
        wait until CLOCK'event and CLOCK = '1';
        if (CONDITION) then
                X <= A;
        else
                wait until CLOCK'event and CLOCK = '1';
        end if;
end process;

-- Явная логика состояния
...
type STATE_TYPE is (SO, S1);
variable STATE : STATE_TYPE;
...
process
begin
        wait until CLOCK'event and CLOCK = '1';
        case STATE is
                when S0 =>
                        if (CONDITION) then
                                X <= A;
                                STATE := S0;   -- STATE устанавливается здесь для избежания
                                               -- петли обратной связи в синтезируемой
                                               -- логике.
                        else
                                STATE := S1;
                        end if;
```

```
                when S1 =>
                        STATE := S0;
        end case;
end process;
```

Примечание : Операторы wait могут использоваться в любом месте процесса , исключая операторы for..loop и подпрограммы. Однако , если какой-либо путь через логику содержит один или более операторов wait , то все пути должны содержать по меньшей мере один оператор wait.

В примере 6-28 показано , как может быть описана схема с синхронным сбросом с помощью оператора **wait** в бесконечном цикле. Сигнал сброса должен проверяться немедленно после каждого оператора **wait** . Операторы присваивания в примере 6-28 (**X <= A;** и **Y <= B;**) представляют собой обыкновенные последовательные операторы , используемые для выполнения этой схемы.

Пример 6-28. Синхронный сброс , использующий операторы wait .

```
process
begin
        RESET_LOOP: loop
                wait until CLOCK'event and CLOCK = '1';
                next RESET_LOOP when (RESET = '1');
                X <= A;
                wait until CLOCK'event and CLOCK = '1';
                next RESET_LOOP when (RESET = '1');
                Y <= B;
        end loop RESET_LOOP;
end process;
```

В примере 6-29 показано два недопустимых использования операторов **wait**. Эти ограничения специфичны для FPGA Express.

Пример 6-29. Недопустимое использование операторов wait .

```
...
type COLOR is (RED, GREEN, BLUE);
attribute ENUM_ENCODING : STRING;
attribute ENUM_ENCODING of COLOR : type is -100 010 001";
signal CLK : COLOR;
...
process
begin
        wait until CLK'event and CLK = RED;
        -- Недопустимо : тип синхроимпульса не кодируется одним битом
        ...
end;
        ...
process
begin
        if (X = Y) then
                wait until CLK'event and CLK = '1';
                ...
        end if;
        -- Недопустимо : не все пути содержат операторы wait
        ...
end;
```

Отличия комбинационных и последовательных процессов

Если в процессе нет операторов **wait** , то он синтезируется с помощью комбинационной логики. Вычисления выполняются в виде немедленного реагирования процесса на изменения во входных сигналах.

Если процесс использует один или более операторов **wait** , то он синтезируется с помощью последовательной логики. При этом вычисления выполняются только один раз для каждого определенного среза синхроимпульса (положительного или отрицательного). Результаты этих вычислений сохраняются в триггерах до тех пор , пока не будет опознан следующий срез.

В триггерах хранятся следующие значения :

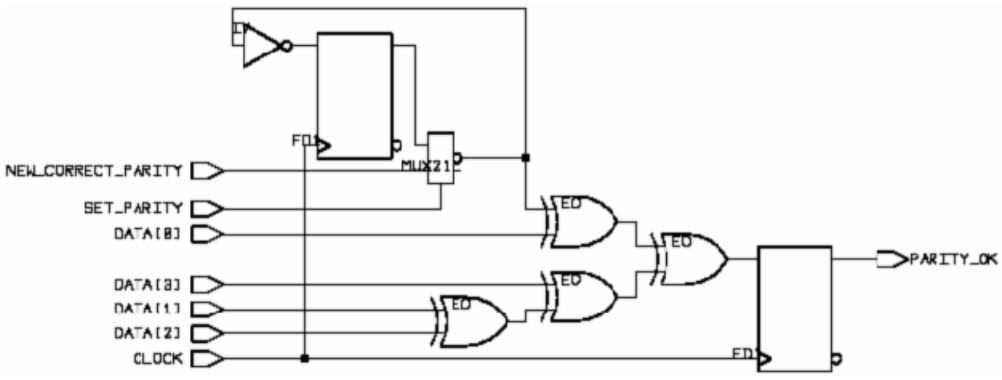
- Сигналы , управляемые процессами ; см. «Оператор присваивания сигнала» в начале этой главы.
- Значения вектора состояния , где вектор состояния может быть явным или неявным (как в примере 6-27).
- Переменные , которые *могут* быть прочитаны перед тем , как установлены.

Примечание : Подобно оператору wait некоторые применения оператора if также могут реализовать синхронную логику , заставляя FPGA Express выполнять регистры или защелки. Эти методы описаны в Главе 8 «Реализация регистров и третьего состояния».

В примере 6-30 оператор **wait** используется для хранения значений в тактируемых циклах. В приведенном фрагменте сравнивается четность значения данных с четностью хранящегося значения. Последнее (**CORRECT_PARITY**) устанавливается из сигнала **NEW_CORRECT_PARITY** , если сигнал **SET_PARITY** является истинным.

Пример 6-30. Проверка на четность с помощью оператора wait .

```
signal CLOCK: BIT;
signal SET_PARITY, PARITY_OK: Boolean ;
signal NEW_CORRECT_PARITY: BIT;
signal DATA: BIT_VECTOR(0 to 3);
...
process
variable CORRECT_PARITY, TEMP: BIT;
        begin
        wait until CLOCK'event and CLOCK = '1';

        -- Если требуется , устанавливается новое правильное значение четности
        if (SET_PARITY) then
                CORRECT_PARITY := NEW_CORRECT_PARITY;
        end if;

        -- Вычисляется четность DATA
        TEMP := '0';
        for I in DATA'range loop
                TEMP := TEMP xor DATA(I);
        end loop;

        -- Сравнивается вычисленная четность с правильным значением
        PARITY_OK <= (TEMP = CORRECT_PARITY);
end process;
```



Заметим , что на синтезированной для примера 6-30 схеме присутствуют два триггера. Первый (входной) триггер хранит значение **CORRECT_PARITY** . Триггер здесь необходим , поскольку **CORRECT_PARITY** читается (когда сравнивается с **TEMP**) перед тем , как устанавливается (если **SET_PARITY** является ложным). Второй (выходной) триггер хранит значение **PARITY_OK** между циклами синхроимпульса. Переменной **TEMP** не дается триггер , поскольку она всегда установлена перед чтением.

Оператор null

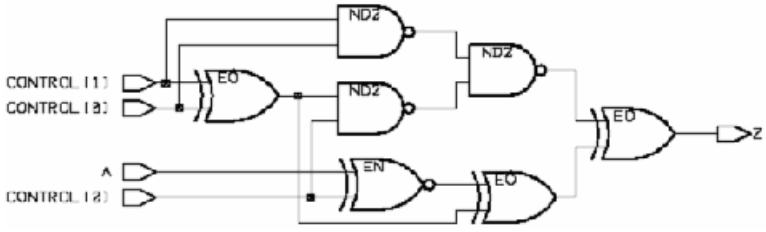
Оператор **null** явно указывает на то , что не нужно производить никакие действия. Оператор **null** часто используется в операторах **case** , поскольку все выборы должны быть перекрыты , даже если некоторые из них игнорируются. Синтаксис этого оператора следующий :

null;

В примере 6-31 показано типичное использование оператора **null**.

Пример 6-31. Оператор null .

```
signal CONTROL: INTEGER range 0 to 7;
signal A, Z: BIT;
...
Z <= A;

case CONTROL is
        when 0 | 7 =>           -- Если от 0 до 7 , то инвертировать A
                Z <= not A;
        when others =>
                null;           -- Если не от 0 до 7 , то ничего не делать
end case;
```



Глава 7. Параллельные операторы

Архитектура VHDL содержит набор параллельных операторов. Каждый параллельный оператор определяет один из взаимосвязанных блоков или процессов , которые описывают общее поведение или структуру проекта. Параллельные операторы в проекте выполняются непрерывно , в отличие от последовательных (см. Главу 6) , которые выполняются один за другим.

Существует два основных параллельных оператора :

оператор процесса

Оператор процесса определяет процесс. Процессы состоят из последовательных операторов (см. Главу 6) , однако сами по себе являются параллельными операторами. Все процессы в проекте выполняются параллельно. Однако , в любой данный момент времени только один последовательный оператор воспринимается внутри каждого процесса. Процесс связан с остальной частью проекта через читаемые и записываемые значения сигналов или портов , объявленных за пределами процесса.

оператор блока

Оператор блока определяет блок. Блоки являются именованным набором параллельных операторов , необязательно использующим локально определенные типы , сигналы , подпрограммы и компоненты.

VHDL обеспечивает две параллельные версии последовательных операторов : параллельные вызовы процедур и параллельные присваивания сигналам. Оператор компонентной реализации ссылается на предварительно определенный аппаратный компонент. Наконец , оператор **generate** создает несколько копий любого последовательного оператора.

Параллельные операторы состоят из

- операторов *process*
- оператора *block*
- параллельных вызовов процедур
- параллельных присваиваний сигналам
- компонентных реализаций
- операторов *generate*

Операторы process

Оператор **process** содержит упорядоченный набор последовательных операторов. Синтаксис этого оператора следующий :

[*label*:] process [(*sensitivity_list*)]
{ *process_declarative_item* }
begin
{ *sequential_statement* }
end process [*label*] ;

Необязательная метка ***label*** именует процесс. Список чувствительности (***sensitivity_list***) является списком всех сигналов (включая порты) , читаемых процессом , в следующем формате :

***signal_name* {, *signal_name*}**

Аппаратное обеспечение , синтезируемое FPGA Express , является чувствительным ко всем сигналам , читаемым процессом. Для уверенности в том , что симулятор VHDL показывает те же результаты , что и синтезируемая схема , список чувствительности процесса должен содержать все сигналы , изменения которых требуют повторной симуляции этого процесса. FPGA Express проверяет списки чувствительности на предмет полноты и выдает предупреждающие сообщения для любых сигналов , которые читаются внутри процесса , но не присутствуют в списке чувствительности. Ошибка будет выдана и в том случае , если тактирующий сигнал читается как данные в процессе.

Примечание : IEEE VHDL не разрешает наличие списка чувствительности , если процесс включает оператор wait.

process_declarative_item объявляет подпрограммы , типы , константы и переменные , локальные для процесса. Эти пункты могут быть любыми из следующих :

- предложение **use**
- объявление подпрограммы
- тело подпрограммы
- объявление типа
- объявление подтипа
- объявление константы
- объявление переменной

Каждый последовательный оператор (***sequential_statement***) описан в Главе 6. Концептуально поведение процесса определяется последовательностью его операторов. После того , как выполнен последний оператор в процессе , работа программы продолжается с первого оператора процесса. Единственное исключение существует при моделировании : если процесс имеет список чувствительности , то он приостанавливается (после выполнения последнего оператора) , пока не произойдет изменение одного из сигналов из списка чувствительности. Если процесс имеет один или более операторов **wait** (и , следовательно , у него нет списка чувствительности) , то он приостанавливается на первом из операторов **wait** , условие ожидания которого является ложным.

Аппаратное обеспечение , синтезируемое для процессе является либо комбинационным (не тактируемым) , либо последовательным (тактируемым). Если процесс включает операторы **wait** или **if*signal*'event** , то его схема содержит последовательные компоненты. Операторы **wait** и **if** описаны в Главе 6.

Примечание : Операторы process обеспечивают натуральные средства для описания концептуально последовательных алгоритмов. Если значения , вычисляемые в процессе , являются безусловно параллельными , рассмотрите использование параллельных операторов присваивания сигналам (см. «Параллельные присваивания сигналам» позже в этой главе).

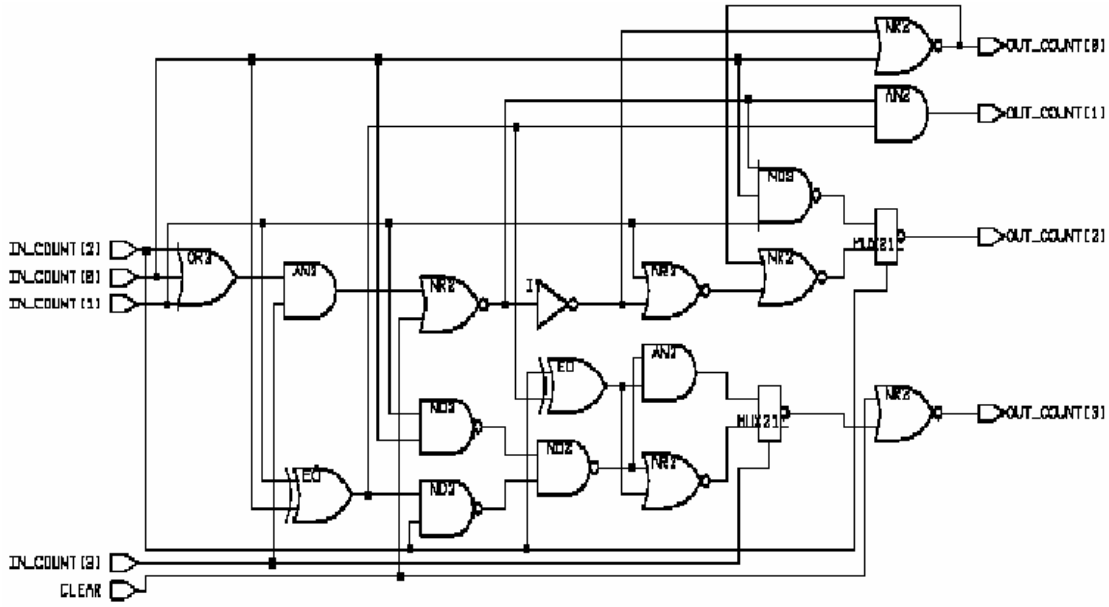
Пример комбинационного процесса

В примере 7-1 показан процесс , который реализует простой счетчик по модулю 10. Рассматриваемый процесс чувствителен (читает) к двум сигналам : **CLEAR** и **IN_COUNT** . Он управляет одним сигналом , **OUT_COUNT** . Если **CLEAR** равен **'1'** или **IN_COUNT** равен **9** , то **OUT_COUNT** устанавливается в ноль. В противном случае **OUT_COUNT** устанавливается на единицу больше , чем **IN_COUNT** .

Пример 7-1. Процесс для счетчика по модулю 10.

```
entity COUNTER is
        port (  CLEAR:      in BIT;
                IN_COUNT:   in INTEGER range 0 to 9;
                OUT_COUNT: out INTEGER range 0 to 9);
        end COUNTER;

architecture EXAMPLE of COUNTER is
begin
        process(IN_COUNT, CLEAR)
        begin
                if (CLEAR = '1' or IN_COUNT = 9) then
                        OUT_COUNT <= 0;
                else
                        OUT_COUNT <= IN_COUNT + 1;
                end if;
        end process;
```

end EXAMPLE;

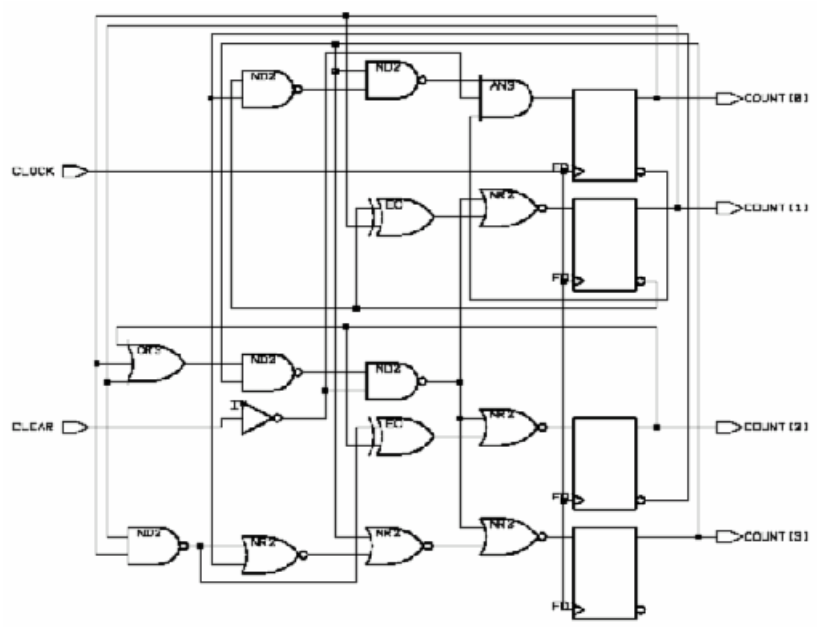


Пример последовательного процесса

Поскольку процесс из примера 7-1 не содержит операторов **wait** , то он синтезируется с помощью комбинационной логики. Альтернативной реализацией счетчика является сохранение его значения внутри процесса с помощью оператора **wait**. В примере 7-2 показана реализация счетчика в виде последовательного (тактируемого) процесса. По каждому переходу **CLOCK** из 0 в 1 , если **CLEAR** равен **'1'** или **COUNT** равен **9** , **COUNT** устанавливается в ноль ; в противном случае **COUNT** инкрементируется на 1.

Пример 7-2. Процесс для счетчика по модулю 10 с оператором wait .

```
entity COUNTER is
        port (  CLEAR:      in BIT;
                CLOCK:      in BIT;
                COUNT:      buffer INTEGER range 0 to 9);
end COUNTER;

architecture EXAMPLE of COUNTER is
begin
        process
        begin
                wait until CLOCK'event and CLOCK = '1';

                if (CLEAR = '1' or COUNT >= 9) then
                        COUNT <= 0;
                else
                        COUNT <= COUNT + 1;
                end if;
        end process;
end EXAMPLE;
```



В примере 7-2 значение переменной **COUNT** хранится в четырех триггерах. Эти триггера генерируются вследствие того , что **COUNT** может быть прочитан до того , как установлен , поэтому его значение должно поставляться из предыдущего синхроцикла. См. «Оператор *wait*» в Главе 6 для более подробной информации.

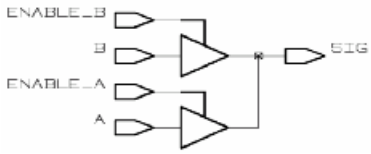
Управляемые сигналы

Если процесс назначает значение сигналу , то он является *драйвером* (источником) этого сигнала. Если более одного процесса или других параллельных операторов управляют сигналом , то этот сигнал имеет *несколько драйверов.*

В примере 7-3 показано два буфера с третьим состоянием , управляющих одним и тем же сигналом (**SIG**). В Главе 8 в разделе «Реализация третьего состояния» показано , как описать технологически независимые приборы с третьим состоянием на VHDL.

Пример 7-3. Сигнал с несколькими драйверами.

```
A_OUT <= A when ENABLE_A else 'Z';
B_OUT <= B when ENABLE_B else 'Z';

process(A_OUT)
begin
        SIG <= A_OUT;
end process;
process(B_OUT)
begin
        SIG <= B_OUT;
end process;
```



Функции шинного разрешения присваивают значение сигналу с несколькими драйверами. См. «Функции разрешения» в разделе «Подпрограммы» Главы 3 для более подробной информации.

Оператор block

Оператор **block** именует набор параллельных операторов. Используйте блоки для иерархической организации параллельных операторов. Синтаксис этого оператора следующий :

***label*: block**
{ *block_declarative_item* }
begin
{ *concurrent_statement* }
end block [*label*];

Обязательный параметр ***label*** именует блок. ***block_declarative_item*** объявляет локальные для блока объекты и может быть любым из следующих пунктов :

- предложение **use**
- объявление подпрограммы
- тело подпрограммы
- объявление типа
- объявление подтипа
- объявление константы
- объявление сигнала
- объявление компонента

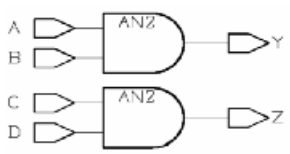
Порядок каждого из параллельных операторов (***concurrent_statement***) в блоке не важен , поскольку каждый из операторов всегда активен.

Примечание : FPGA Express не поддерживает защищенные блоки.

Объекты , объявленные в блоке , являются видимыми для него самого и для всех вложенных в него блоков. Когда вложенный блок объявляет объект с тем же именем , что и в родительском блоке , то объявление вложенного блока переписывает объявление родительского (внутри вложенного блока). В примере 7-4 показано использование вложенных блоков.

Пример 7-4. Вложенные блоки.

```
B1: block
        signal S: BIT;          -- Объявление "S" в блоке B1
begin
        S <= A and B;           -- "S" из B1

        B2: block
                signal S: BIT;  -- Объявление "S" в блоке B2
        begin
                S <= C and D;   -- "S" из B2

                B3: block
                begin
                        Z <= S; -- "S" из B2
                end block B3;
        end block B2;

        Y <= S;                 -- "S" из B1
end block B1;
```



Параллельные вызовы процедур

Параллельный вызов процедуры является вызовом процедуры , который используется как параллельный оператор ; он применяется в архитектуре или блоке чаще , чем в процессе. Параллельный вызов процедуры эквивалентен процессу , содержащему один последовательный вызов процедуры. Синтаксис этого вызова аналогичен синтаксису последовательного вызова процедуры :

procedure_name [([name =>] expression
{ , [name =>] expression })] ;

Эквивалентный процесс чувствителен ко всем **in** и **inout** параметрам процедуры. В примере 7-5 показано объявление процедуры , которая затем вызывается параллельно , и эквивалентный этому процесс.

Пример 7-5. Параллельный вызов процедуры и эквивалентный процесс.

```
procedure ADD( signal A, B: in BIT;
               signal SUM: out BIT);
...
ADD(A, B, SUM);        -- Параллельный вызов процедуры
...
process(A, B)          -- Эквивалентный процесс
begin
    ADD(A, B, SUM);    -- Последовательный вызов процедуры
end process;
```

FPGA Express реализует вызовы процедур и функций с помощью логики до тех пор , пока вы не используете директиву компилятора **map_to_entity** (см. «Размещение подпрограмм в компоненты (объекты) в Главе 6).

Наиболее общим использованием параллельных вызовов процедур является получение нескольких копий процедуры. Например , допустим , что класс сигналов **BIT_VECTOR** должен содержать только один бит со значением **1** , а остальные - со значением **0**. Предположим , что у вас есть несколько сигналов различной ширины , которые вы хотите отслеживать одновременно. Одним из подходов является написание процедуры для обнаружения ошибки в сигнале **BIT_VECTOR** , а затем параллельный вызов ее для каждого сигнала.

В примере 7-6 показана процедура **CHECK** , которая определяет , действительно ли битовый вектор содержит только один элемент со значением **'1'**; в противном случае **CHECK** устанавливает свой **out** параметр **ERROR** в **TRUE** .

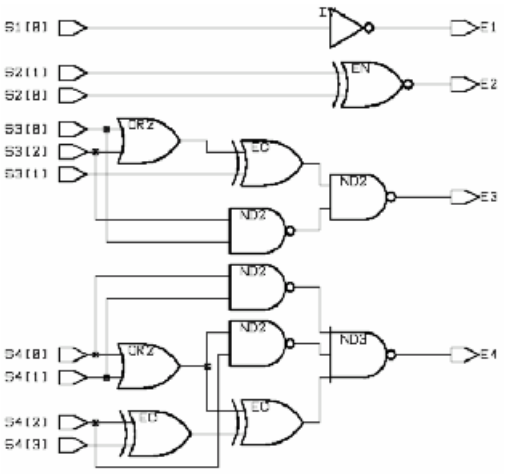
Пример 7-6. Определение процедуры для примера 7-7.

```
procedure CHECK(    signal A:          in BIT_VECTOR;
                    signal ERROR:   out Boolean ) is
    variable FOUND_ONE: Boolean := FALSE;
                                    -- Устанавливает TRUE , когда встречает '1'
begin
    for I in A'range loop                       -- Цикл по всем битам вектора
        if A(I) = '1' then                      -- Нашли '1'
            if FOUND_ONE then                   -- Мы уже находили единицу ?
                ERROR <= TRUE;                  -- Нашли две '1'
                return;                         -- Завершение процедуры
            end if;
            FOUND_ONE := TRUE;                  -- Запоминаем , что мы видели '1'
        end if;
    end loop;
    ERROR <= not FOUND_ONE; -- Будет ошибка , если единица не найдена
end;
```

В примере 7-7 показано , как процедура **CHECK** вызывается параллельно для четырех сигналов с различными размерами битовых векторов.

Пример 7-7. Параллельные вызовы процедур.

```
BLK: block
        signal S1: BIT_VECTOR(0 to 0);
        signal S2: BIT_VECTOR(0 to 1);
        signal S3: BIT_VECTOR(0 to 2);
        signal S4: BIT_VECTOR(0 to 3);
        signal E1, E2, E3, E4: Boolean ;

begin
        CHECK(S1, E1);          -- Параллельный вызов процедуры
        CHECK(S2, E2);
        CHECK(S3, E3);
        CHECK(S4, E4);
end block BLK;
```



Параллельные присваивания сигналам

Параллельное присваивание сигналу эквивалентно процессу , содержащему такое последовательное присваивание. Таким образом , каждое параллельное присваивание сигналу определяет для него новый драйвер. Простейшая форма параллельного присваивания сигнала имеет следующий вид:

***target* <= *expression*;**

где ***target*** является сигналом , который принимает значение выражения ***expression***. В примере 7-8 показано , как значение выражения **A and B** параллельно присваивается сигналу **Z**.

Пример 7-8. Параллельное присваивание сигналу.

```
BLK: block
signal A, B, Z: BIT;
begin
        Z <= A and B;
end block BLK;
```

Двумя другими формами параллельного присваивания сигналу является условное и выборочное присваивание.

Условное присваивание сигналу

Другой формой параллельного присваивания сигналу является условное присваивание. Оно имеет следующий синтаксис :

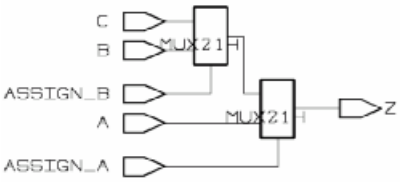
target **<=** ***{ expression*** **when** ***condition*** **else** ***}***
expression;

где ***target*** - сигнал , который принимает значение выражения ***expression***. Используемое выражение ***expression*** является первым , для которого Булевское условие ***condition*** является истинным. Когда выполняется оператор условного присваивания сигналу , каждое условие ***condition*** тестируется по порядку. Первое условие , которое принимает истинное значение , приводит к назначению выражения ***expression*** сигналу ***target***. Если ни одно из условий не является истинным , сигналу назначается последнее выражение. Если истинными являются два и более условий , то эффективным оказывается только первое , аналогично первому истинному ветвлению в операторе **if**.

В примере 7-9 показано условное присваивание сигналу **Z**. Этому сигналу присваивается значение одного из трех сигналов : **A** , **B** или **C**. Источник зависит от значения выражений **ASSIGN_A** и **ASSIGN_B.** Заметим , что присваивание **A** имеет приоритет над присваиванием **B** , а присваивание **B** имеет приоритет над присваиванием **C** , так как присваиванием управляет первое из истинных условий.

Пример 7-9. Условное присваивание сигналу.

```
Z <= A when ASSIGN_A = '1' else
     B when ASSIGN_B = '1' else
     C;
```



В примере 7-10 показан процесс , эквивалентный условному присваиванию сигналу из примера 7-9.

Пример 7-10. Процесс , эквивалентный условному присваиванию сигналу.

```
process(A, ASSIGN_A, B, ASSIGN_B, C)
begin
        if ASSIGN_A = '1' then
                Z <= A;
        elsif ASSIGN_B = '1' then
                Z <= B;
        else
                Z <= C;
        end if;
end process;
```

Выборочное присваивание сигналу

Последним типом параллельного присваивания сигналу является выборочное присваивание , которое имеет следующий синтаксис :

with ***choice_expression*** **select**
target **<=** ***{ expression*** **when** ***choices, }***
expression **when** ***choices*****;**

где ***target*** - сигнал , который принимает значение выражения ***expression***. Выбираемое выражение ***expression*** является первым из тех , чей выбор ***choices*** включает значение ***choice_expression***. Синтаксис ***choices*** такой же , как и для оператора **case** :

choice **{ |** ***choice*** **}**

Каждый выбор может быть либо статическим выражением (таким , как **3**) , либо статическим диапазоном (таким , как **1 to 3**). Тип ***choice_expression*** определяет тип каждого выбора. Каждое значение из диапазона ***choice_expression*** должно покрываться одним выбором ***choice***. Последний ***choice*** может быть **others** , при этом согласовываются все оставшиеся (невыбранные) значения из диапазона выражения ***choice_expression***. Выбор **others** , если таковой присутствует , согласовывает ***choice_expression*** только в том случае , если ни один из других выборов не совпадает.

Оператор **with..select** вычисляет выражение ***choice_expression*** и сравнивает это значение со значением каждого выбора ***choice***. Предложение **when** совпавшего значения ***choice*** присваивает значение своего выражения ***expression*** сигналу ***target***.

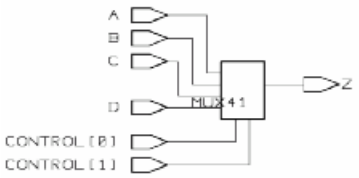
На выборы накладываются следующие ограничения :

- Никакие два выбора не могут перекрываться.
- Если выбор **others** не присутствует , то все возможные значения ***choice_expression*** должны быть перекрыты соответствующим набором выборов.

В примере 7-11 показан сигнал **Z** , присваиваемый из **A** , **B** , **C** или **D**. Назначение зависит от текущего значения **CONTROL** .

Пример 7-11. Выборочное присваивание сигналу.

```
signal A, B, C, D, Z: BIT;
signal CONTROL: bit_vector(1 down to 0);
. . .
with CONTROL select
        Z  <=  A when "00",
               B when "01",
               C when "10",
               D when "11";
```



В примере 7-12 показан процесс , эквивалентный выборочному присваиванию сигналу из примера 7-11.

Пример 7-12. Процесс , эквивалентный выборочному присваиванию сигналу.

```
process(CONTROL, A, B, C, D)
begin
```

```
        case CONTROL is
                when 0 =>
                        Z <= A;
                when 1 =>
                        Z <= B;
                when 2 =>
                        Z <= C;
                when 3 =>
                        Z <= D;
        end case;
end process;
```

Компонентные реализации

Компонентная реализация относится к предварительно определенному аппаратному компоненту текущего проекта на текущем уровне иерархии. Вы можете использовать компонентные реализации для определения иерархии проекта. Вы также можете использовать элементы , не записанные на VHDL , такие как компоненты из технологической библиотеки FPGA , элементы , определенные на языке аппаратного описания Verilog , или из основной технологической библиотеки. Операторы компонентной реализации могут использоваться для построения списков цепей в VHDL.

Оператор компонентной реализации показывает :

- Имя данной реализации компонента.
- Имя компонента для включения в текущий объект.
- Метод связи для портов компонента.

Синтаксис этого оператора следующий :

```
instance_name : component_name port map (
                [ port_name => ] expression
                , [ port_name => ] expression } );
```

где ***instance_name*** именует реализацию компонента типа ***component_name.*** **port map** (карта портов) соединяет каждый порт данной реализации ***component_name*** с сигнально значимым выражением ***expression*** в текущем объекте. Значение выражения ***expression*** может быть именем сигнала , индексным именем , скользящим именем или множеством. Если ***expression*** является зарезервированным VHDL словом **open** , то соответствующий порт остается несвязанным.

Вы можете размещать порты по сигналам с помощью именованной или позиционной записи. Вы можете включать как именованные , так и позиционные связи в карту портов , но все позиционные связи должны располагаться перед именованными.

Примечание : При именованной связи имена портов компонента должны точно согласовываться с объявленными именами портов компонента. При позиционной связи действительные выражения для портов должны располагаться в том же порядке , как объявлены порты компонента.

Пример 7-13 показывает объявление компонента (2-входовой элемент И-НЕ) и следующие за ним три эквивалентных оператора компонентной реализации.

Пример 7-13. Объявление и реализации компонента.

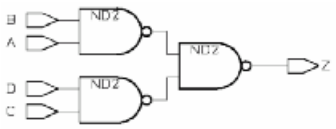
```
component ND2
        port(A, B: in BIT; C: out BIT);
end component;
. . .
signal X, Y, Z: BIT;
. . .
U1: ND2 port map(X, Y, Z);                    -- позиционная
U2: ND2 port map(A => X, C => Z, B => Y);  -- именованная
```

U3: ND2 port map(X, Y, C => Z); -- смешанная

В примере 7-14 показан оператор реализации компонента , определяющий простой список цепей. Три реализации U1 , U2 и U3 являются реализациями 2-входового элемента И-НЕ , объявленного в примере 7-13.

Пример 7-14. Простой список цепей.

```
signal TEMP_1, TEMP_2: BIT;
. . .
U1: ND2 port map(A, B, TEMP_1);
U2: ND2 port map(C, D, TEMP_2);
U3: ND2 port map(TEMP_1, TEMP_2, Z);
```



Операторы generate

Оператор **generate** создает ноль или более копий включенного набора параллельных операторов. Существует два типа оператора **generate** :

for... generate
количество копий определяется дискретным диапазоном
if... generate
ноль или более копий делается условно

Оператор for .. generate

Этот оператор имеет следующий синтаксис :

***label*: for *identifier* in *range* generate**
{ *concurrent_statement* }
end generate [*label*] ;

Обязательный параметр ***label*** именует этот оператор (пригоден для вложенных операторов **generate**). Использование идентификатора ***identifier*** в этой конструкции аналогично оператору **for..loop** :

- ***identifier*** не объявляется где-либо. Он объявляется автоматически самим оператором **generate** и является полностью локальным для цикла. Идентификатор цикла переписывает любой другой идентификатор с таким же именем , но только внутри цикла.
- Значение идентификатора может быть прочитано только внутри цикла , однако вы не можете присвоить ему значение. Кроме того , значение идентификатора не может быть присвоено какому-либо параметру , режим которого **out** или **inout** .

FPGA Express требует , чтобы диапазон ***range*** был *вычисляемым* целым диапазоном в любом из следующих видов :

integer_expression* to *integer_expression
integer_expression* downto *integer_expression

Каждое выражение ***integer_expression*** вычисляет целое значение.

Параллельный оператор ***concurrent_statement*** может быть любым из операторов , описанных в этой главе , включая другие операторы **generate**.

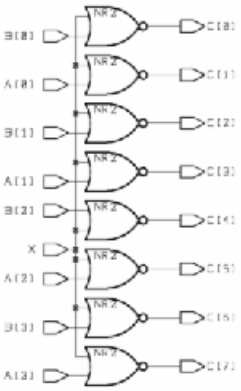
Оператор **for..generate** выполняется следующим образом :

1. Объявляется новая локальная целая переменная с именем ***identifier***.
2. Идентификатору ***identifier*** присваивается первое значение из диапазона ***range*** , и каждый параллельный оператор выполняется один раз.
3. Идентификатору назначается следующее значение из диапазона , и каждый параллельный оператор выполняется еще раз.
4. Шаг 3 повторяется , пока идентификатору не будет присвоено последнее значение из диапазона. Каждый параллельный оператор затем выполняется в последний раз , и выполнение программы продолжается , начиная с оператора , стоящего вслед за **end generate** . Идентификатор цикла стирается.

В примере 7-15 показан фрагмент программы , которая комбинирует и перемежает два четырехбитовых массива **A** и **B** в восьмибитовый массив **C**.

Пример 7-15. Оператор for..generate .

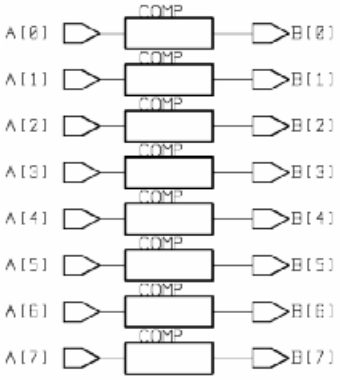
```
signal A, B : bit_vector(3 downto 0);
signal C : bit_vector(7 downto 0);
signal X : bit;
. . .
GEN_LABEL: for I in 3 downto 0 generate
        C(2*I + 1) <= A(I) nor X;
        C(2*I) <= B(I) nor X;
end generate GEN_LABEL;
```



Наиболее общим применением оператора **generate** является создание нескольких копий компонентов , процессов или блоков. Пример 7-16 демонстрирует такое использование для компонентов. В примере 7-17 показано , как генерировать несколько копий процесса. В примере 7-16 задействован атрибут массива VHDL **'range** , используемый с оператором **for..generate** для реализации набора компонентов **COMP** , которые связывают соответствующие элементы битовых векторов **A** и **B**.

Пример 7-16. Оператор for..generate , работающий над целым массивом.

```
component COMP
port (  X : in bit;
        Y : out bit);
end component;
. . .
signal A, B: BIT_VECTOR(0 to 7);
. . .
GEN: for I in A'range generate
        U: COMP port map (X => A(I) , Y => B(I));
end generate GEN;
```



Неограниченные массивы и атрибуты массивов описаны в разделе «Типы массив» Главы 4. Атрибуты массивов показаны в примере 4-9.

Оператор if . . generate

Синтаксис этого оператора следующий :

***label*: if *expression* generate**
{ *concurrent_statement* }
end generate [*label*] ;

где ***label*** идентифицирует (именует) этот оператор. Выражение ***expression*** является любым выражением , которое вычисляет Булевское значение. Параллельный оператор ***concurrent_statement*** является любым из операторов , описанных в этой главе , включая другие операторы **generate**.

Примечание : В отличие от оператора if , описанного в Главе 6 , оператор if..generate не имеет каких-либо ветвлений else или elsif.

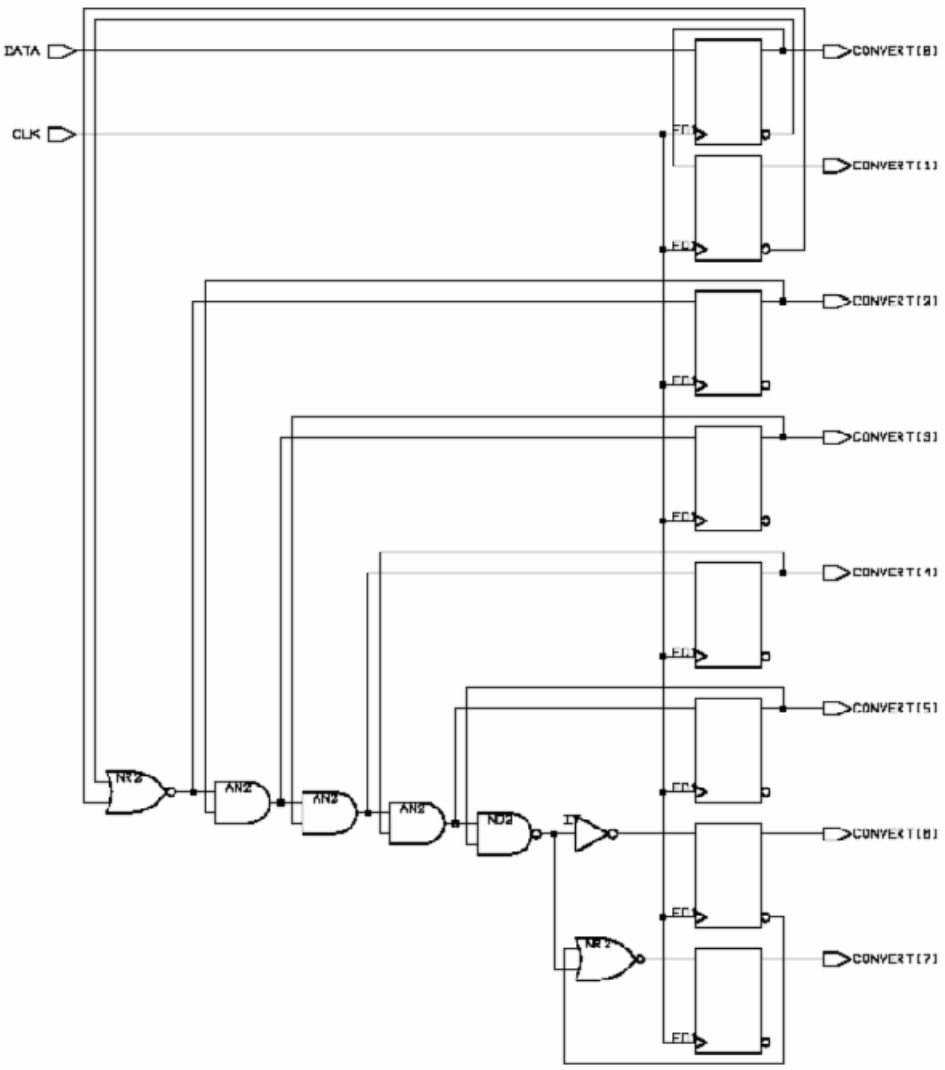
Вы можете использовать оператор **if..generate** для генерации регулярной структуры , которая имеет разную схему на своих концах. Оператор **for..generate** применяется для итерации по нужной ширине проекта , а набор операторов **if..generate -** для определения начального , среднего и конечного набора связей.

В примере 7-17 показано технологически независимое описание *N*-битового последовательно-парллельного преобразователя. Данные в *N*-битовом буфере тактируются справа налево. В каждом синхроцикле каждый бит в *N*-битовом буфере сдвигается вверх на один бит , а входящий бит **DATA** переносится в младший бит буфера.

Пример 7-17. Типичное использование оператора if..generate .

```
entity CONVERTER is
        generic(N: INTEGER := 8);

        port(   CLK, DATA: in BIT;
                CONVERT: buffer BIT_VECTOR(N-1 downto 0));
end CONVERTER;

architecture BEHAVIOR of CONVERTER is
        signal S : BIT_VECTOR(CONVERT'range);
begin
        G: for I in CONVERT'range generate

                G1:     -- Сдвигает (N-1) бит данных в старший бит
                        if (I = CONVERT'left) generate
                        process begin
                                wait until (CLK'event and CLK = '1');
                                CONVERT(I) <= S(I-1);
```

```
                end process;
        end generate G1;
        G2:     -- Сдвигает средние биты вверх
                if (I > CONVERT'right and I < CONVERT'left) generate
                S(I) <= S(I-1) and CONVERT(I);
                process begin
                        wait until (CLK'event and CLK = '1');
                        CONVERT(I) <= S(I-1);
                end process;
        end generate G2;
        G3:     -- Переносит DATA в младший бит
                if (I = CONVERT'right) generate
                process begin
                        wait until (CLK'event and CLK = '1');
                        CONVERT(I) <= DATA;
                end process;
                S(I) <= CONVERT(I);
        end generateG3;

    end generateG;
end BEHAVIOR;
```



Глава 8. Реализация регистров и третьего состояния

В общем вы можете использовать несколько различных , но логически эквивалентных описаний VHDL для описания схемы. Для написания программ на VHDL , которые синтезируются в наиболее эффективные схемы , обсудим следующие темы :

- реализация регистров
- реализация третьего состояния

Вы можете использовать VHDL для того , чтобы сделать ваш проект более эффективным в терминах размеров синтезируемой схемы и скорости работы , следующим образом :

- Проект , которому нужны некоторые , но не все из его переменных или сигналов , хранящихся в процессе работы , может быть записан таким образом , чтобы минимизировать количество необходимых для него защелок и триггеров.
- Проект , который наиболее легко описывается с помощью нескольких уровней иерархии , может быть синтезирован более эффективно , если часть иерархии будет свернута в процессе синтеза.

Реализация регистров

FPGA Express обеспечивает реализацию регистров с помощью операторов **wait** и **if**. *Регистр* является простым однобитовым прибором памяти - либо триггером , либо защелкой. Триггер является прибором памяти , тактируемым по срезу. Защелка же чувствительна к уровню сигнала.

Используйте оператор **wait** для выполнения триггеров в синтезируемой схеме. FPGA Express создает триггера для всех сигналов и некоторых присваиваемых значений переменных в процессе с оператором **wait**. Оператор **if** может использоваться для выполнения регистров (триггеров или защелок) для сигналов и переменных в ветвлениях оператора **if**.

Для применения регистров , описывающих защелки и триггера , и изучения их эффективного использования вам необходимо познакомиться со следующими понятиями :

- использование реализаций регистров
- описание защелок
- описание триггеров
- эффективное использование регистров

Использование реализаций регистров

Использование реализаций регистров включает описание тактирующих сигналов и использование операторов **wait** и **if** . Необходимо обсудить также рекомендуемые модели для различных типов реализуемых регистров и ограничения Synopsys.

Описание тактирующих сигналов

FPGA Express может реализовать асинхронные элементы памяти из программ VHDL , написанных в натуральном стиле. Используйте операторы **wait** и **if** для тестирования растущего или падающего среза сигнала.

Наиболее общее использование имеет следующий вид :

```
process
begin
        wait until ( edge);
        ...
end process;
...

process ( sensitivity_list)
begin
        if ( edge)
```

```
        ...
        end if;
end process;
```

Другая форма :

```
process ( sensitivity_list)
begin
        if (...) then
            ...
        elsif (...)
            ...
        elsif ( edge) then
            ...
        end if;
end process;
```

где ***edge*** ссылается на выражение , которое тестирует положительный или отрицательный срез сигнала. Синтаксис выражения ***edge*** следующий :

```
SIGNAL'event          and SIGNAL = '1'      -- растущий срез
NOT SIGNAL'stable     and SIGNAL = '1'      -- растущий срез

SIGNAL'event          and SIGNAL = '0'      -- падающий срез
NOT SIGNAL'stable     and SIGNAL = '0'      -- падающий срез
```

В операторе **wait** выражение ***edge*** также может быть

```
signal = '1'      -- растущий срез
signal = '0'      -- падающий срез
```

Выражение ***edge*** должно быть единственным условием оператора **if** или **elsif**. У вас может быть только одно выражение ***edge*** в операторе **if** , и оператор **if** не должен иметь предложения **else**. Выражение ***edge*** не может быть ни частью другого логического выражения , ни использоваться как аргумент.

```
if ( edge and RST = '1') -- Неправильное использование ; edge должно быть только условием

Any_function( edge);     -- Неправильное использование ; edge не может быть аргументом

if X > 5 then
        sequential_statement;
elsif edge then
        sequential_statement;
else
        sequential_statement;
end if;                  -- Неправильное использование ; не используйте edge как
                         -- промежуточное выражение.
```

Эти примеры иллюстрируют три некорректных использования выражения ***edge***. В первом из них выражение ***edge*** является частью большого Булевского выражения. Во втором выражение ***edge*** используется как аргумент. В третьем выражение ***edge*** используется как промежуточное условие.

Отличия операторов *wait* и *if*

Иногда вы можете использовать операторы **wait** и **if** как взаимозаменяемые. Оператор **if** обычно является предпочтительным , поскольку он обеспечивает лучший контроль над особенностями реализуемых регистров , как описано в следующем разделе.

IEEE VHDL требует , чтобы в процессе с оператором **wait** не было списка чувствительности.

Оператор **if *edge*** может появиться в любом месте процесса. Список чувствительности процесса должен содержать все сигналы , читаемые процессом , включая сигнал ***edge***. Вообще говоря применяются следующие руководящие принципы :

- Синхронные процессы (процессы , которые вычисляют значения только по тактирующим срезам) должны быть чувствительными к тактирующему сигналу.
- Асинхронные процессы (процессы , которые вычисляют значения по тактирующим срезам и когда асинхронные условия становятся истинными) должны быть чувствительными к тактирующему сигналу (если таковой имеется) и ко входам , которые воздействуют на асинхронное поведение.

Рекомендуемое использование возможностей реализуемых регистров

Реализуемые регистры могут поддерживать стили описания , отличные от тех , которые здесь описаны. Тем не менее , для получения наилучших результатов :

- Ограничивайте каждый процесс одним типом реализуемых элементов памяти : защелка , защелка с асинхронным сбросом или установкой , триггер , триггер с асинхронным сбросом или триггер с синхронной установкой.
- Используйте следующие шаблоны :

```
LATCH:          process( sensitivity_list)
                begin
                        if LATCH_ENABLE then
                                ...
                        end if;
                end process;

LATCH_ASYNC_SET:
                        ...
attribute async_set_reset of SET : signal is "true";
                        ...
                process( sensitivity_list)
                begin
                        if SET then
                                Q <= '1';
                        elsif LATCH_ENABLE then
                                ...
                        end if;
                end process;

FF:     process(CLK)
                begin
                        if edge then
                                ...
                        end if;
                end process;

FF_ASYNC_RESET:  process(RESET, CLK)
                        begin
                                if RESET then
                                        Q <= '0';
                                elsif edge then
                                        Q <= ...;
                                end if;
```

```
                    end process;

FF_SYNC_RESET:      process(RESET, CLK)
                    begin
                            if edge then
                                    if RESET then
                                            Q <= '0';
                                    else
                                            Q <= ...;
                                    end if;
                            end if;
                    end process;
```

Примеры этих шаблонов приведены в разделах «Описание защелок» и «Описание триггеров» позже в этой главе.

Ограничения возможностей регистров

Не используйте более одного выражения **if *edge*** в процессе.

```
process(CLK_A, CLK_B)
begin
        if(CLK_A'event and CLK_A = '1') then
                A <= B;
        end if;
        if(CLK_B'event and CLK_B = '1') then          --Неправильно
                C <= B;
        end if;
end process;
```

Не присваивайте значение переменной или сигналу по ложному ветвлению оператора **if *edge***. Такое присваивание эквивалентно проверке *отсутствия* тактирующего среза , которая не имеет адекватной аппаратной реализации.

```
process(CLK)
begin
        if(CLK'event and CLK = '1') then
                SIG <= B;
        else
                SIG <= C;       -- Неправильно
        end if;
end process;
```

Если переменной присваивается значение внутри конструкции ***edge*** , не читайте эту переменную позже в этом же процессе.

```
process(CLK)
variable EDGE_VAR, ANY_VAR: BIT;

begin
        if (CLK'event and CLK = '1') then
                EDGE_SIGNAL <= X;
                EDGE_VAR := Y;
                ANY_VAR := EDGE_VAR;     -- Правильно
        end if;

        ANY_VAR := EDGE_VAR;             -- Неправильно
end process;
```

Не используйте выражение ***edge*** как операнд.

if not(CLK'event and CLK = '1') then – Неправильно

Задержки в регистрах

Если вы используете спецификации задержки для значений , которые могут быть реализованы через регистры , то моделирование их поведения будет отличаться от логически синтезируемого посредством FPGA Express. Например , описание в примере 8-1 содержит информацию о задержке , которая заставляет FPGA Express синтезировать схему с неожиданным поведением.

Пример 8-1. Задержки в регистрах.

```
component flip_flop (    D, clock: in BIT;
                         Q: out BIT;);
end component;

process ( A, C, D, clock );
        signal B: BIT;
begin
        B <= A after 100ns;

        F1: flip_flop port map ( A, C, clock ),
        F2: flip_flop port map ( B, D, clock );
end process;
```

В примере 8-1 **B** изменяется на 100 наносекунд позже , чем **A**. Если период тактирующих импульсов меньше 100 нс , то при моделировании схемы выход **D** окажется на один или более синхроимпульсов позади выхода **C .** Тем не менее , поскольку FPGA Express игнорирует информацию о задержке , **A** и **B** изменяют свои значения в одно и то же время , и то же самое происходит с **C** и **D**. Такое поведение *отличается* от моделируемой схемы.

Когда вы используете информацию о задержке в своих проектах , убедитесь в том , что эти задержки не оказывают воздействия на значения , хранящиеся в регистрах. Вообще вы можете осторожно включать информацию о задержках в ваше описание , если при этом не изменяются значения в тактируемых триггерах.

Описание защелок

FPGA Express реализует защелки из не полностью определенных условных выражений. В примере 8-2 оператор **if** выполняется в виде защелки , так как у него нет предложения **else** :

Пример 8-2. Реализация защелки.

```
process(GATE, DATA)
begin
        if (GATE = '1') then
                Q <= DATA;
        end if;
end process;
```

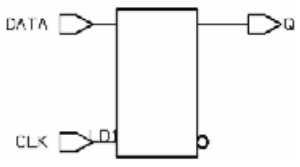


Рис.8-1. Реализация защелки.

Выполненная защелка использует **CLK** как тактирующий вход и **DATA** как вход данных (см. пример 8-2).

Автоматическая реализация защелок

Сигнал или переменная , которые не управляются всеми условиями , становятся защелкиваемым значением. Как показано в примере 8-3 , **TEMP** становится защелкиваемым значением , поскольку он назначается только когда **PHI** равно **1**.

Пример 8-3. Автоматическая реализация защелки.

```
if(PHI = '1') then
        TEMP <= A;
end if;
```

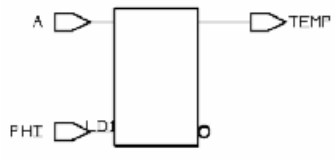


Рис.8-2. Автоматическая реализация защелки.

Чтобы избежать реализации защелок , назначайте значение сигналу при всех возможных условиях , как показано в примере 8-4.

Пример 8-4. Полностью определенный сигнал : защелка не выполняется.

```
if (PHI = '1') then
        TEMP <= A;
else
        TEMP <= '0';
end if;
```

Ограничения возможностей реализуемых защелок

Вы не можете прочитать условно присвоенную переменную после оператора **if** , в котором произошло это присваивание. Условно присваиваемой переменной новое значение присваивается при некоторых , но не всех возможных условиях. Таким образом , переменная всегда должна иметь значение перед тем , как она будет прочитана.

```
signal X, Y: BIT;
. . .
process
variable VALUE: BIT;
begin
        if ( condition) then
                VALUE := X;
        end if;
        Y <= VALUE;             -- Неправильно
end;
```

При моделировании реализация защелок выполняется , поскольку сигналы и переменные могут хранить значение некоторое время. Сигнал или переменная хранят свое значение до тех пор , пока оно не будет переназначено. FPGA Express вставляет защелку в аппаратное обеспечение для дублирования этого хранимого состояния.

Переменные , объявленные локально внутри подпрограммы , не хранят свое значение на протяжении времени. Каждый раз при вызове подпрограммы ее переменные инициализируются заново. Таким образом , FPGA Express не выполняет защелки для переменных , объявленных в подпрограммах. В примере 8-5 защелки не реализуются.

Пример 8-5. Функция без реализации защелок.

```
function MY_FUNC(DATA, GATE : BIT) return BIT is
variable STATE: BIT;

begin
        if GATE then
                STATE := DATA;
        end if;

        return STATE;
end;
. . .
Q <= MY_FUNC(DATA, GATE);
```

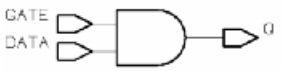


Рис.8-3. Функция без реализации защелки.

Пример — проект с двухфазными синхроимпульсами

С помощью применения возможностей реализации защелок вы можете описать сетевые структуры , такие как двухфазные системы , технологически независимым способом. В примере 8-6 показана простая двухфазная система с тактирующими сигналами **PHI_1** и **PHI_2** .

Пример 8-6. Двухфазные синхроимпульсы.

```
entity LATCH_VHDL is
        port(   PHI_1, PHI_2, A : in BIT;
                t: out BIT);
end LATCH_VHDL;

architecture EXAMPLE of LATCH_VHDL is
        signal TEMP, LOOP_BACK: BIT;
begin
        process(PHI_1, A, LOOP_BACK)
        begin
                if(PHI_1 = '1') then
                        TEMP <= A and LOOP_BACK;
                end if;
        end process;

        process(PHI_2, TEMP)
        begin
                if(PHI_2 = '1') then
                        LOOP_BACK <= not TEMP;
                end if;
        end process;
```

```
        t <= LOOP_BACK;

end EXAMPLE;
```

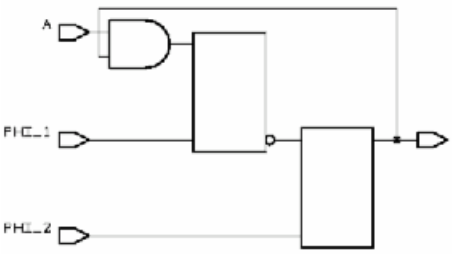


Рис.8-4. Двухфазные синхроимпульсы.

FPGA Express не выполняет автоматически двухфазные защелки (приборы с главными и подчиненными синхроимпульсами). Для использования таких приборов вам необходимо выполнить их как компоненты (см. описание в Главе 3).

Описание триггеров

В примере 8-7 показано , как конструкция ***edge*** создает триггер.

Пример 8-7. Реализация триггера.

```
process(CLK, DATA)
begin
        if (CLK'event and CLK = '1') then
                Q <= DATA;
        end if;
end process;
```

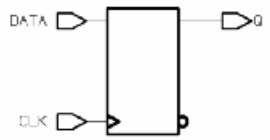


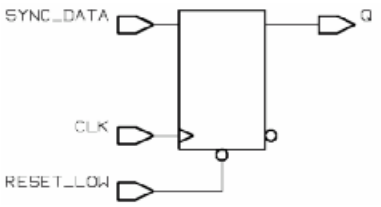
Рис.8-5. Реализация триггера.

Триггер с асинхронным сбросом

В примере 8-8 показано , как определить триггер с асинхронным сбросом.

Пример 8-8. Реализация триггера с асинхронным сбросом.

```
process(RESET_LOW, CLK, SYNC_DATA)
begin
        if RESET_LOW = '0' then
                Q <= '0';
        elsif (CLK'event and CLK = '1') then
                Q <= SYNC_DATA;
        end if;
end process;
```



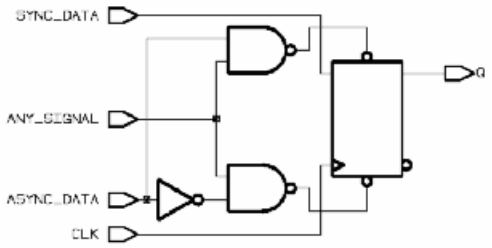
Отметим , как соединен триггер в примере 8-8 :

- Вход **D** триггера соединен с **SYNC_DATA** .
- Если условие сброса является вычисляемым (см. «Вычисляемые операнды» в Главе 5) , то вывод **SET** или **CLEAR** триггера соединяется с сигналом **RESET** (или **RESET_LOW**).
- Если условие сброса (**ANY_SIGNAL** в примере 8–9) является не вычисляемым , то **SET** соединяется с **(ANY_SIGNAL AND ASYNC_DATA)** и **CLEAR** соединяется с **(ANY_SIGNAL AND NOT(ASYNC_DATA))** , как показано в примере 8–9.

В примере 8-9 показана реализация триггера с асинхронным сбросом , у которого условие сброса является не вычисляемым.

Пример 8-9. Реализация триггера с асинхронным сбросом или установкой.

```
process (CLK, ANY_SIGNAL, ASYNC_DATA, SYNC_DATA)
begin
        if (ANY_SIGNAL) then
                Q <= ASYNC_DATA;
        elsif (CLK'event and CLK = '1') then
                Q <= SYNC_DATA;
        end if;
end process;
```



Пример — синхронный проект с асинхронным сбросом

В примере 8-10 описан синхронный конечный автомат (FSM) с асинхронным сбросом.

Пример 8-10. Синхронный конечный автомат с асинхронным сбросом.

```
package MY_TYPES is
        type STATE_TYPE is (S0, S1, S2, S3);
end MY_TYPES;

use WORK.MY_TYPES.ALL;

entity STATE_MACHINE is
        port(   CLK, INC, A, B:         in BIT;
                RESET:                  in Boolean ;
                t:                      out BIT);
end STATE_MACHINE;

architecture EXAMPLE of STATE_MACHINE is
        signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
begin
        SYNC: process(CLK, RESET)
```

```
        begin
                if (RESET) then
                        CURRENT_STATE <= S0;
                elsif (CLK'event and CLK = '1') then
                        CURRENT_STATE <= NEXT_STATE;
                end if;
        end process SYNC;

        FSM: process(CURRENT_STATE, A, B)
        begin
                t <= A;                 -- Назначение по умолчанию
                NEXT_STATE <= S0;  -- Назначение по умолчанию

                if (INC = '1') then
                        case CURRENT_STATE is
                                when S0 =>
                                        NEXT_STATE <= S1;
                                when S1 =>
                                        NEXT_STATE <= S2;
                                        t <= B;
                                when S2 =>
                                        NEXT_STATE <= S3;
                                when S3 =>
                                        null;
                        end case;
                end if;
        end process FSM;
end EXAMPLE;
```

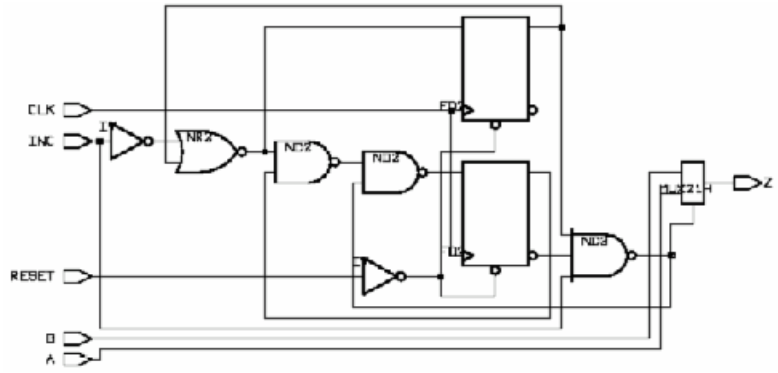


Рис.8-6. Синхронный конечный автомат с асинхронным сбросом.

Атрибуты

В данном разделе обсуждаются новые атрибуты , используемые для помощи в реализации регистров. Атрибуты , объявленные в библиотеке VHDL , называются блоком объявлений атрибутов Synopsys.

```
attribute async_set_reset : string;
attribute sync_set_reset : string;
attribute async_set_reset_local : string;
attribute sync_set_reset_local : string;
attribute async_set_reset_local_all : string;
attribute sync_set_reset_local_all : string;
attribute one_hot : string;
attribute one_cold : string;
```

async_set_reset

Атрибут **async_set_reset** присоединяется к однобитовым сигналам с помощью своей конструкции. FPGA Express проверяет сигналы с установленным **TRUE** атрибутом **async_set_reset** для определения того , не влияют ли эти сигналы на асинхронную установку или сброс защелок во всем проекте.

Синтаксис **async_set_reset** следующий :

attribute async_set_reset of signal_name,. : signal is "true";

Защелка с асинхронными входами установки или сброса

Асинхронный сигнал сброса для защелки выполняется путем подачи **0** на вывод "Q" защелки. Асинхронный сигнал установки выполняется , соответственно , путем подачи **1** на вывод "Q" . Хотя FPGA Express не требует , чтобы сброс (установка) был первым условием в вашем ветвлении , лучше писать программу VHDL именно таким образом.

В примере 8-11 показано , как определить защелку с асинхронным входом сброса. Для определения защелки с асинхронной установкой измените логику , как обозначено в комментариях.

Пример 8-11. Реализация защелки с асинхронным входом сброса.

```
attribute async_set_reset of clear : signal is
"true";
process(clear, gate, a)
begin
        if ( clear = ’1’) then
                q <= ’0’;
        elsif (gate = ’1’) then
                q <= a;
        end if;
end process;
```

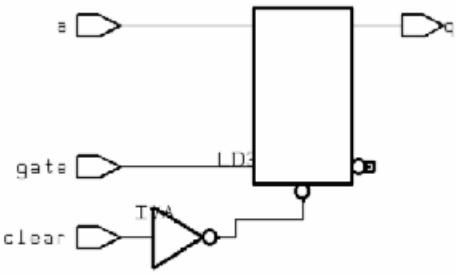


Рис.8-7. Реализация защелки с асинхронным сбросом.

sync_set_reset

Атрибут **sync_set_reset** присоединяется к однобитовым сигналам с помощью своей конструкции. FPGA Express проверяет сигналы с установленным **TRUE** атрибутом **sync_set_reset** для определения того , не влияют ли эти сигналы на синхронную установку или сброс триггеров во всем проекте.

Синтаксис **sync_set_reset** следующий :

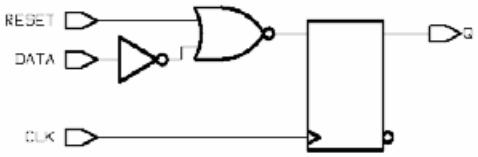
attribute sync_set_reset of signal_name,... : signal is "true";

Триггер с синхронным входом сброса

В примере 8-12 показано , как определить триггер с синхронным сбросом.

Пример 8-12. Реализация триггера с синхронным входом сброса.

```
attribute sync_set_reset of RESET, SET : signal is
"true";
process(RESET, CLK)
begin
        if (CLK'event and CLK = '1') then
                if RESET = '1' then
                        Q <= '0';
                else
                        Q <= DATA_A;
                end if;
        end if;
end process;

process (SET, CLK)
begin
        if (CLK'event and CLK = '1') then
                if SET = '1' then
                        T <= '1';
                else
                        T <= DATA_B;
                end if;
        end if;
end process;
```



async_set_reset_local

Атрибут **async_set_reset_local** присоединяется к метке процесса со значением из списка однобитовых сигналов , заключенного в двойные кавычки. Каждый сигнал в списке трактуется , как если бы он имел атрибут **async_set_reset** , прикрепленный к определенному процессу. Синтаксис **async_set_reset_local** следующий :

```
attribute async_set_reset_local of process_label :
label is
        "signal_name,...";
```

Пример 8-13. Асинхронный сброс/установка одиночного блока.

```
library IEEE;
library synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity e_async_set_reset_local is
        port(   reset, set, gate: in std_logic;
                y, t: out std_logic);
end e_async_set_reset_local;
architecture rtl of e_async_set_reset_local is
attribute async_set_reset_local of direct_set_reset
: label
        is "reset, set";
begin
```

```
        direct_set_reset: process (reset, set)
        begin
                if (reset = '1') then
                        y <= '0';          -- асинхронный сброс
                elsif (set = '1') then
                        y <= '1';          -- асинхронная установка
                end if;
        end process direct_set_reset;

        gated_data: process (gate, reset, set)
        begin
                if (gate = '1') then
                        if (reset = '1') then
                                t <= '0';          -- пропускаемые данные
                        elsif (set = '1') then
                                t <= '1';          -- пропускаемые данные
                        end if;
                end if;
        end process gated_set_reset;

end rtl;
```

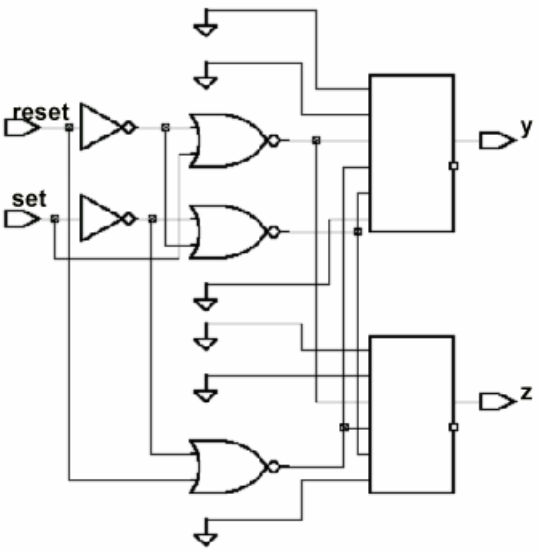


Рис.8-8. Асинхронный сброс/установка одиночного блока.

sync_set_reset_local

Атрибут **sync_set_reset_local** присоединяется к метке процесса со значением из списка одно-битовых сигналов , заключенного в двойные кавычки. Каждый сигнал в списке трактуется , как если бы он имел атрибут **sync_set_reset** , прикрепленный к определенному процессу. Синтаксис **sync_set_reset_local** следующий :

```
attribute sync_set_reset_local of process_label :
label is "signal_name,..."
```

Пример 8-14. Синхронный сброс/установка одиночного блока.

```
library IEEE;
library synopsys;
```

```
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity e_sync_set_reset_local is
        port(clk, reset, set, gate : in std_logic; y, t: out std_logic);
end e_sync_set_reset_local;

architecture rtl of e_sync_set_reset_local is
attribute sync_set_reset_local of clocked_set_reset : label is "reset, set";
begin
        clocked_reset: process (clk, reset, set)
        begin
                if (clk'event and clk = '1') then
                        if (reset = '1') then
                                y <= '0';          -- синхронный сброс
                        else
                                y <= '1';          -- синхронная установка
                        end if;
                end if;
        end process clocked_set_reset;

        gated_data: process (clk, gate, reset, set)
        begin
                if (clk'event and clk = '1') then
                        if (gate = '1') then
                                if (reset = '1') then
                                        t <= '0';          -- пропускаемые данные
                                elsif (set = '1') then
                                        t <= '1';          -- пропускаемые данные
                                end if;
                        end if;
                end if;
        end process gated_set_reset;

end rtl;
```

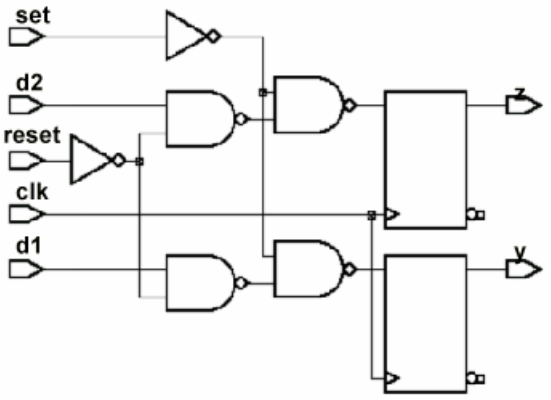


Рис.8-9. Синхронный сброс/установка одиночного блока.

async_set_reset_local_all

Атрибут **async_set_reset_local_all** присоединяется к метке процесса. Этот атрибут определяет , что все сигналы в процессе используются для детектирования условия асинхронного сброса или установки , чтобы выполнить защелки или триггера. Синтаксис **async_set_reset_local_all** следующий :

```
attribute async_set_reset_local_all of process_
label,... : label is "true";
```

Пример 8-15. Асинхронный сброс/установка части проекта.

```
library IEEE;
library synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity e_async_set_reset_local_all is
        port(reset, set, gate, gate2: in std_logic; y, t, w: out std_logic);
end e_async_set_reset_local_all;

architecture rtl of e_async_set_reset_local_all is
attribute async_set_reset_local_all of
          direct_set_reset, direct_set_reset_too: label is "true";
begin
        direct_set_reset: process (reset, set)
        begin
                if (reset = '1') then
                        y <= '0';          -- асинхронный сброс
                elsif (set = '1') then
                        y <= '1';          -- асинхронная установка
                end if;
        end process direct_set_reset;
        direct_set_reset_too: process (gate, reset, set)
        begin
                if (gate = '1') then
                        if (reset = '1') then
                                t <= '0';          -- асинхронный сброс
                        elsif (set = '1') then
                                t <= '1';          -- асинхронная установка
                        end if;
                end if;
        end process direct_set_reset_too;
        gated_data: process (gate2, reset, set)
        begin
                if (gate = '1') then
                        if (reset = '1') then
                                w <= '0';          -- пропускаемые данные
                        elsif (set = '1') then
                                w <= '1';          -- пропускаемые данные
                        end if;
                end if;
        end process gated_set_reset;
end rtl;
```

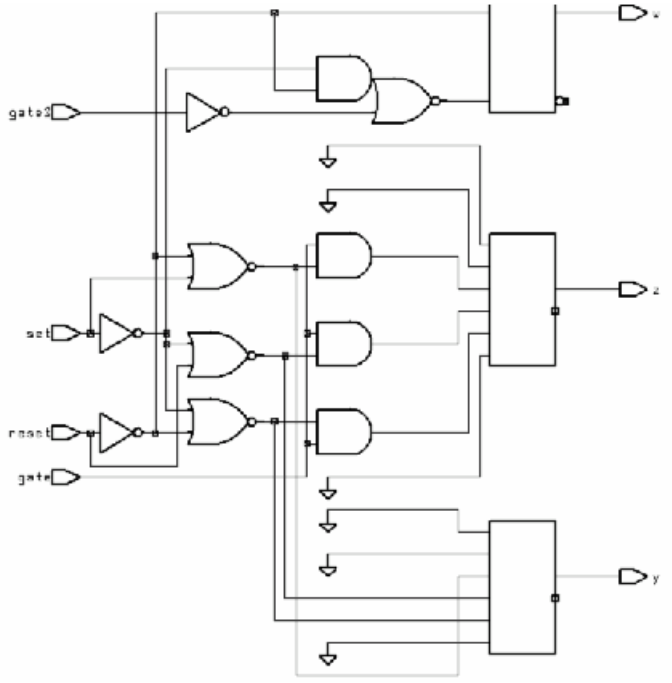


Рис.8-10. Асинхронный сброс/установка части проекта.

sync_set_reset_local_all

Атрибут **sync_set_reset_local_all** присоединяется к метке процесса. Этот атрибут определяет , что все сигналы в процессе используются для детектирования условия синхронного сброса или установки , чтобы выполнить защелки или триггера. Синтаксис **sync_set_reset_local_all** следующий :

```
attribute sync_set_reset_local_all of process_label,... : label is "true";
```

Пример 8-16. Синхронный сброс/установка части проекта.

```
library IEEE;
library synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity e_sync_set_reset_local_all is
        port(clk, reset, set, gate, gate2: in std_logic; y, t, w: out std_logic);
end e_sync_set_reset_local_all;

architecture rtl of e_sync_set_reset_local_all is
attribute sync_set_reset_local_all of
          clocked_set_reset, clocked_set_reset_too: label is "true";
begin
        clocked_set_reset: process (clk, reset, set)
        begin
                if (clk'event and clk = '1') then
                        if (reset = '1') then
                                y <= '0';          -- синхронный сброс
                        elsif (set = '1') then
                                y <= '1';          -- синхронная установка
                        end if;
                end if;
        end process clocked_set_reset;
        clocked_set_reset_too: process (clk, gate, reset, set)
```

```
        begin
                if (clk'event and clk = '1') then
                        if (gate = '1') then
                                if (reset = '1') then
                                        t <= '0';          -- синхронный сброс
                                elsif (set = '1') then
                                        t <= '1';          -- синхронная установка
                                end if;
                        end if;
                end if;
        end process clocked_set_reset_too;

        gated_data: process (clk, gate2, reset, set)
        begin
                if (clk'event and clk = '1') then
                        if (gate = '1') then
                                if (reset = '1') then
                                        w <= '0';          -- пропускаемые данные
                                elsif (set = '1') then
                                        w <= '1';          -- пропускаемые данные
                                end if;
                        end if;
                end if;
        end process gated_set_reset;

end rtl;
```

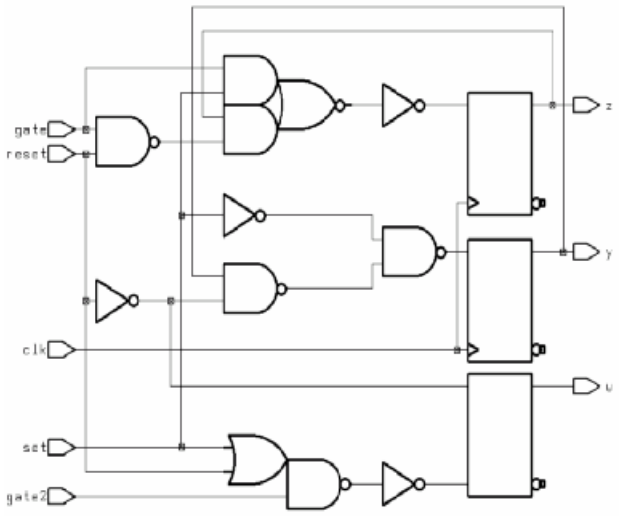


Рис.8-11. Синхронный сброс/установка части проекта.

Примечание : Используйте директивы one_hot и one_cold для выполнения D-триггеров с асинхронными сигналами сброса и установки. Эти два атрибута уведомляют FPGA Express , что только один из объектов в списке является активным в данный момент времени. Если вы определяете активные высокие сигналы , используйте one_hot. Для активных низких используйте one_cold. Каждый атрибут имеет два определенных объекта.

one_hot

Директива **one_hot** берет один аргумент из списка сигналов в двойных кавычках , разделенных запятыми. Этот атрибут показывает , что группа сигналов является **one_hot** , или другими словами , в любой момент времени не более , чем один сигнал может иметь значение логической **1**. Вы должны быть уверены в том , что группа сигналов действительно является **one_hot** («горячей едини-

цей»). FPGA Express не генерирует какую-либо логику для проверки этого утверждения. Синтаксис **one_hot** следующий :

```
attribute one_hot signal_name,... : label is "true";
```

Пример 8-17. Использование one_hot для сброса и установки.

```
library IEEE;
library synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity e_one_hot is
        port(reset, set, reset2, set2: in std_logic; y, t: out std_logic);
attribute async_set_reset of reset, set : signal is "true";
attribute async_set_reset of reset2, set2 : signal is "true";
attribute one_hot of reset, set : signal is "true";
end e_one_hot;

architecture rtl of e_one_hot is
begin
        direct_set_reset: process (reset, set )
        begin
                if (reset = '1') then
                        y <= '0';          -- асинхронный сброс "reset"
                elsif (set = '1') then
                        y <= '1';          -- асинхронная установка "set"
                end if;
        end process direct_set_reset;

        direct_set_reset_too: process (reset2, set2 )
        begin
                if (reset2 = '1') then
                        t <= '0';          -- асинхронный сброс "reset2"
                elsif (set2 = '1') then
                        t <= '1';          -- асинхронная установка "set2"
                end if;
        end process direct_set_reset_too;

        -- синтез synopsys выключается
        process (reset, set)
        begin
                assert not (reset='1' and set='1')
                        report "One-hot violation"
                        severity Error;
        end process;
        -- синтез synopsys включается

end rtl;
```

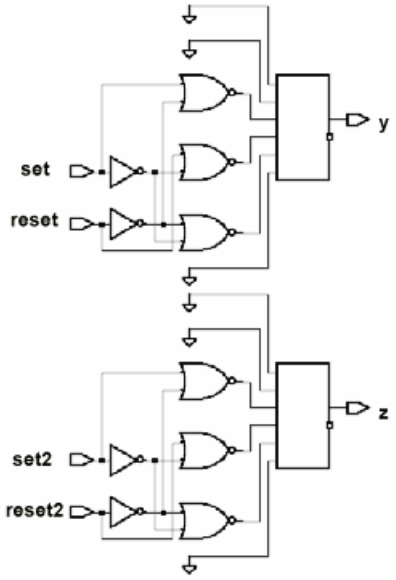


Рис.8-12. Использование ***one_hot*** для сброса и установки.

one_cold

Директива **one_cold** аналогична директиве **one_hot**. **one_cold** показывает , что не более , чем один сигнал в группе может иметь значение логического **0** в любой момент времени. Синтаксис **one_cold** следующий :

```
attribute one_cold signal_name,... : label is "true";
```

Пример 8-18. Использование ***one_cold*** для сброса и установки.

```
library IEEE;
library synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;
entity e_one_cold is
        port(reset, set, reset2, set2: in std_logic; y, t: out std_logic);
attribute async_set_reset of reset, set : signal is "true";
attribute async_set_reset of reset2, set2 : signal is "true";
attribute one_cold of reset, set : signal is "true";
end e_one_cold;

architecture rtl of e_one_cold is
begin

        direct_set_reset: process (reset, set )
        begin
                if (reset = '0') then
                        y <= '0';          -- асинхронный сброс "not reset"
                elsif (set = '0') then
                        y <= '1';          -- асинхронная установка "not set"
                end if;
        end process direct_set_reset;
        direct_set_reset_too: process (reset2, set2 )
        begin
                if (reset2 = '0') then
                        t <= '0';          -- асинхронный сброс "not reset2"
                elsif (set2 = '0') then
                        t <= '1';          -- асинхронная установка "(not reset2) (not set2)"
```

```
            end if;
        end process direct_set_reset_too;
        -- синтез synopsys выключается
        process (reset, set)
        begin
            assert not (reset='0' and set='0')
                report "One-cold violation"
                severity Error;
        end process;
        -- синтез synopsys включается
end rtl;
```

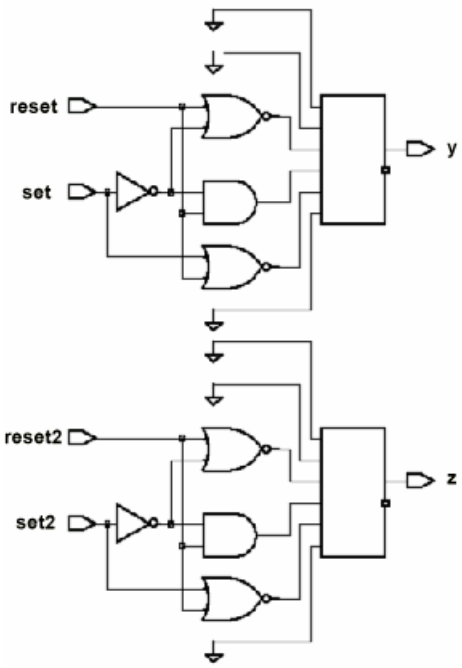


Рис.8-13. Использование ***one_cold*** для сброса и установки.

Реализация защелок и триггеров в FPGA Express

FPGA Express выполняет защелки и триггера следующим образом :

- Триггера с асинхронным сбросом
  FPGA Express сообщает условия асинхронного сброса или установки триггеров.
- Защелки с асинхронным сбросом
  FPGA Express интерпретирует каждый объект , управляющий защелкой , как синхронный. Если вы хотите асинхронно сбросить или установить защелку , установите эту переменную в **TRUE** .
- Триггера с обратной связью
  FPGA Express удаляет все петли обратной связи у триггеров. Например , будет удалена петля обратной связи , получающаяся из такого оператора , как **Q=Q**. При удалении устойчивой обратной связи из простого D-триггера он становится синхронно загружаемым.
- Триггера с инверсной обратной связью
  FPGA Express удаляет все петли инверсной обратной связи у триггеров. Например , будет удалена петля обратной связи , получающаяся из такого оператора , как **Q=Q** , и синтезирован T-триггер.
- Отчет о выполненных модулях
  FPGA Express генерирует краткий отчет о реализованных защелках , триггерах или приборах с третьим состоянием.

Организуйте ваше HDL описание так , чтобы при его реализации получилось минимально необходимое количество триггеров. В примере 8-19 показано описание , в котором реализуется слишком много триггеров.

Пример 8-19. Схема с шестью выполненными триггерами.

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

entity ex8_13 is
        port (    clk , reset : in std_logic;
                  and_bits , or_bits , xor_bits : out std_logic);
end ex8_13;

architecture rtl of ex8_13 is
begin
        process
        variable count : std_logic_vector (2 downto 0);
        begin
                wait until (clk'event and clk = '1');
                        if (reset = '1') then
                                count := "000";
                        else count := count + 1;
                        end if;
                and_bits <= count(2) and count(1) and count(0);
                or_bits <= count(2) or count(1) or count(0);
                xor_bits <= count(2) xor count(1) xor count(0);
        end process;
end rtl;
```

В примере 8-19 выходы **AND_BITS** , **OR_BITS** и **XOR_BITS** зависят исключительно от значения **COUNT** . Поскольку **COUNT** является регистровой величиной , то эти три выхода не нужно делать регистровыми. Чтобы избежать реализации лишних регистров , назначайте выходы изнутри процесса , у которого нет оператора **wait**.

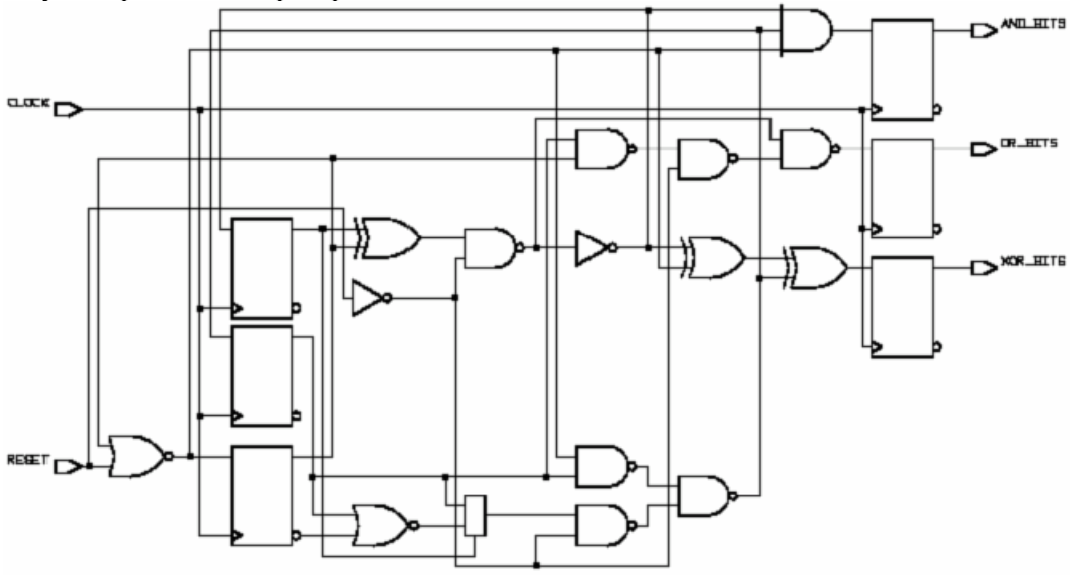


Рис.8-14. Схема с шестью выполненными регистрами.

В примере 8-20 показано описание с двумя процессами , одно с оператором **wait** и одно без него. Стиль описания позволяет вам выбрать , какие сигналы делать регистровыми , а какие - нет.

Пример 8-20. Схема с тремя выполненными регистрами.

```
use work.ARITHMETIC.all;
entity COUNT is
        port(CLOCK, RESET: in BIT;
        AND_BITS, OR_BITS, XOR_BITS : out BIT);
end COUNT;

architecture RTL of COUNT is
        signal COUNT : UNSIGNED (2 downto 0);
begin

        REG: process                -- Регистровая логика
        begin
                wait until CLOCK'event and CLOCK = '1';
                if (RESET = '1') then
                        COUNT <= "000";
                else
                        COUNT <= COUNT + 1;
                end if;
        end process;

        COMBIN: process(COUNT)      – Комбинационная логика
        begin
                AND_BITS <= COUNT(2) and COUNT(1) and COUNT(0);
                OR_BITS <= COUNT(2) or COUNT(1) or COUNT(0);
                XOR_BITS <= COUNT(2) xor COUNT(1) xor COUNT(0);
        end process;

end RTL;
```

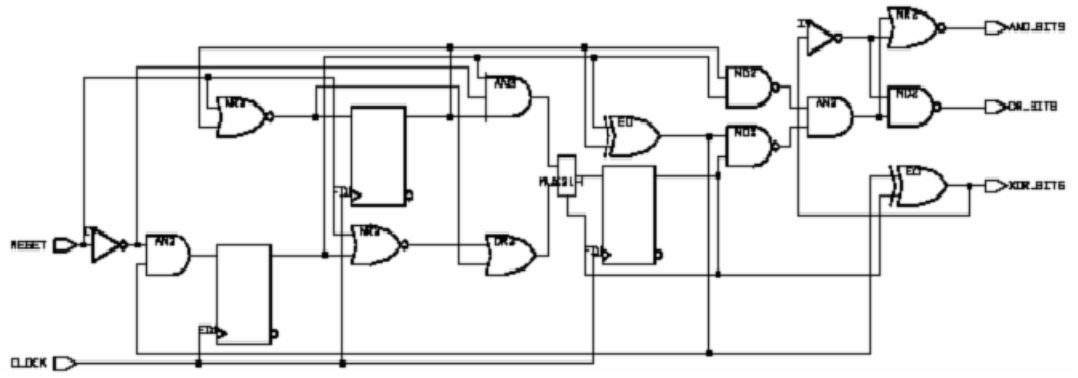


Рис.8-15. Схема с тремя выполненными регистрами.

Такая техника разделения комбинационной логики от регистровой или последовательной логики полезна при описании конечных автоматов (см. примеры в Приложении А).

Пример — использование синхронных и асинхронных процессов

Вам может потребоваться хранить некоторые значения , вычисленные в процессе , в триггерах , в то время как изменение других значений будет разрешено между срезами синхроимпульсов. Такую процедуру можно осуществить , разбив алгоритм между двумя процессами, один из которых будет содержать оператор **wait** . Поместим регистровые (синхронные) присваивания в этот процесс. Поместим другие (асинхронные) присваивания в другой процесс (без оператора **wait**). Для связи ме-

жду двумя процессами используйте сигналы. Например , предположим , что вы хотите построить проект со следующими характеристиками :

- Входы **A_1** , **A_2** , **A_3** и **A_4** изменяются асинхронно.
- Выход **t** управляется одним из сигналов **A_1** , **A_2** , **A_3** или **A_4**.
- Вход **CONTROL** доступен только по положительному срезу **CLOCK** . Значение на срезе определяет , какой из четырех входов будет выбран в течение следующего синхроцикла.
- Выход **t** всегда должен отражать изменения значения выбранного в настоящий момент сигнала.

Реализация такого алгоритма потребует два процесса. Процесс с оператором **wait** синхронизирует значение **CONTROL**. Другой процесс мультиплексирует выход в зависимости от синхронизированного управления. Сигнал **SYNC_CONTROL** связывает два процесса. В примере 8-21 показан текст программы и схема одной из возможных реализаций.

Пример 8-21. Два процесса - синхронный и асинхронный.

```
entity SYNC_ASYNC is
        port (  CLOCK: in BIT;
                CONTROL: in INTEGER range 0 to 3;
                A: in BIT_VECTOR(0 to 3);
                t: out BIT);
end SYNC_ASYNC;

architecture EXAMPLE of SYNC_ASYNC is
        signal SYNC_CONTROL: INTEGER range 0 to 3;
begin

        process
        begin
                wait until CLOCK'event and CLOCK = '1';
                SYNC_CONTROL <= CONTROL;
        end process;

        process (A, SYNC_CONTROL)
        begin
                t <= A(SYNC_CONTROL);
        end process;

end EXAMPLE;
```

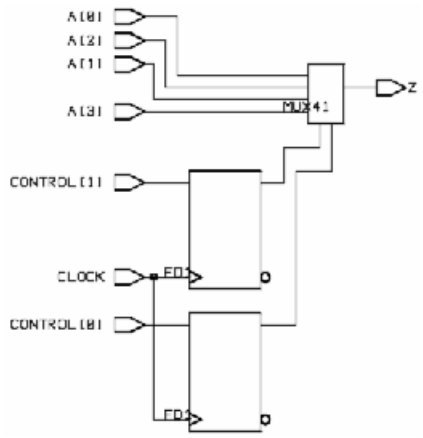


Рис.8-16. Два процесса - синхронный и асинхронный.

Реализация третьего состояния

FPGA Express может выполнить элементы с третьим состоянием (высокоимпедансный выход) через кодирование перечисления на VHDL. После выполнения реализации FPGA Express размещает элементы в определенной технологической библиотеке. См. «Кодирование перечисления» в Главе 4 для более подробной информации.

Когда переменной присваивается значение **'Z'** , выход элемента с третьим состоянием становится недоступным. В примере 8-22 показан фрагмент VHDL для элемента с третьим состоянием.

Пример 8-22. Создание элемента с третьим состоянием в VHDL.

```
signal OUT_VAL, IN_VAL: std_logic;
...
if (COND) then
        OUT_VAL <= IN_VAL;
else
        OUT_VAL <= 'Z';       -- присваивается высокий импеданс
end if;
```

Вы можете присвоить высокоимпедансное значение четырехбитовой шине с помощью **"ZZZZ"** . Из одиночного процесса реализуется один прибор с третьим состоянием. В примере 8-23 показан именно такой случай.

Пример 8-23. Выполнение одного прибора с третьим состоянием из одиночного процесса.

```
process (sela, a, selb, b) begin
        t <= 'z';
        if (sela = '1') then
                t <= a;
        if (selb = '1') then
                t <= b;
end process;
```

В примере 8-24 выполняются два прибора с третьим состоянием.

Пример 8-24. Выполнение двух приборов с третьим состоянием.

```
process (sela, a) begin
        if (sela = '1') then
                t = a;
        else t = 'z';
end process;

process (selb, b) begin
        if (selb = '1') then
                t = b;
        else t = 'z';
end process;
```

Для реализации третьего состояния также может использоваться условное присваивание VHDL.

Присваивание значения *Z*

Присваивание переменным значения **Z** допустимо. Значение **Z** также может появляться в вызовах функций , операторах возврата и множествах. Однако , за исключением сравнения с **Z ,** вы не можете использовать это значение в выражениях. В примере 8-25 показано некорректное использование **Z** (в выражении) , а в примере 8-26 - корректное использование **Z** (в сравнении).

Пример 8-25. Некорректное использование значения Z в выражении.

```
OUT_VAL <= 'Z' and IN_VAL;
...
```

Пример 8-26. Корректное выражение сравнения с Z.

```
if IN_VAL = 'Z' then
...
```

Предупреждение : Выражения сравнения с Z синтезируются , как если бы значения были не равны Z.

Например :

```
if X = 'Z' then
...
```

синтезируется как :

```
if FALSE then
...
```

Если вы используете выражения , в которых значения сравниваются с **'Z' ,** то результаты моделирования до и после синтеза могут отличаться. По этой причине FPGA Express выдает предупреждение при синтезе таких сравнений.

Защелкивающиеся переменные с третьим состоянием

Если переменная защелкивается в том же процессе , в котором она принимает третье состояние, то разрешение третьего состояния **Z** также защелкивается. Этот процесс показан в примере 8-27.

Пример 8-27. Третье состояние , реализуемое с разрешением записи в регистр.

```
-- Создается триггер на вход и на разрешение
if (THREESTATE = '0') then
        OUTPUT <= 'Z';
elsif (CLK'event and CLK = '1') then
        if (CONDITION) then
                OUTPUT <= INPUT;
        end if;
end if;
```

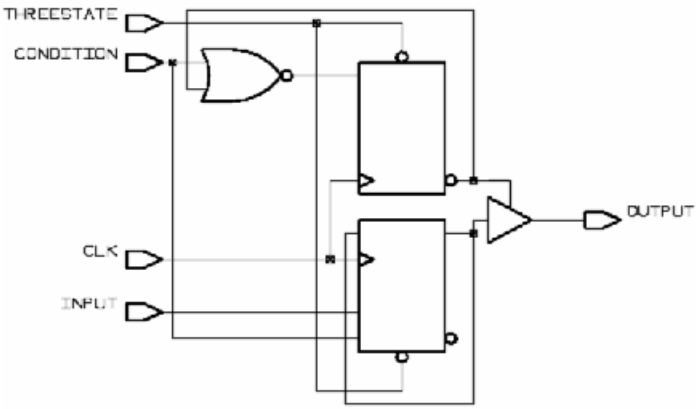


Рис.8-17. Третье состояние , реализуемое с разрешением записи в регистр.

В примере 8-27 элемент с третьим состоянием имеет регистровый сигнал разрешения. Пример 8-28 использует два процесса для реализации третьего состояния с триггером только на входе.

Пример 8-28. Защелкиваемое третье состояние с триггером на входе.

```
entity LATCH_3S is
        port(   CLK, THREESTATE, INPUT: in std_logic;
                OUTPUT: out std_logic; CONDITION: in Boolean );
end LATCH_3S;

architecture EXAMPLE of LATCH_3S is
        signal TEMP: std_logic;
begin

        process(CLK, CONDITION, INPUT)
        begin                                           -- creates three-state
                if (CLK'event and CLK = '1') then
                        if (CONDITION) then
                                TEMP <= INPUT;
                        end if;
                end if;
        end process;

        process(THREESTATE, TEMP)
        begin
                if (THREESTATE = '0') then
                        OUTPUT <= 'Z';
                else
                        OUTPUT <= TEMP;
                end if;
        end process;
end EXAMPLE;
```

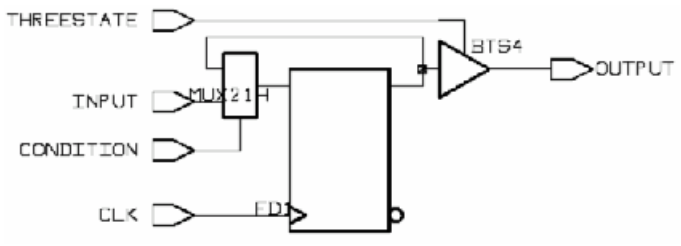


Рис.8-18. Защелкиваемое третье состояние с триггером на входе.

Глава 9. Директивы FPGA Express

В Synopsys определены различные методы , обеспечивающие схемную информацию проекта непосредственно в вашем исходном тексте VHDL.

- Используя директивы FPGA Express , вы можете осуществлять прямую трансляцию из VHDL в компоненты со специальными комментариями. Эти комментарии включают или выключают трансляцию , определяют один из нескольких методов аппаратного разрешения и обеспечивают нужное размещение подпрограмм в аппаратные компоненты.
- Используя Synopsys-определенные VHDL атрибуты , вы можете добавить относящийся к синтезу сигнал и ограничительную информацию для портов , компонентов и объектов. Эта информация используется FPGA Express в процессе синтеза.

Для ознакомления с директивами FPGA Express обсудим следующие темы :

- Нотация для директив FPGA Express
- Директивы FPGA Express

- Атрибуты и ограничения синтеза

Нотация для директив FPGA Express

Директивы FPGA Express являются специальными комментариями VHDL (*синтетические комментарии*) , которые воздействуют на работу FPGA Express. Эти комментарии просто являются специальным случаем обычных комментариев VHDL , поэтому они игнорируются другими средствами VHDL. Синтетические комментарии используются только для работы непосредственно FPGA Express.

Синтетические комментарии начинаются с двух дефисов (**--**) так же , как и обычные комментарии. Если вслед за этими символами стоит слово **pragma** или **synopsys** , то оставшаяся часть комментария интерпретируется FPGA Express как директива.

Примечание : FPGA Express выдает синтаксическую ошибку , если встречает неизвестную директиву после слов -- synopsys или -- pragma.

Директивы FPGA Express

Существует три типа директив :

- Директивы запуска и останова трансляции
  - **-- pragma translate_off**
  - **-- pragma translate_on**
  - **-- pragma synthesis_off**
  - **-- pragma synthesis_on**
- Директивы функции разрешения
  - **-- pragma resolution_method wired_and**
  - **-- pragma resolution_method wired_or**
  - **-- pragma resolution_method three_state**
- Директивы импликации компонента
  - **-- pragma map_to_entity *entity_name***
  - **-- pragma return_port_name *port_name***

Другие директивы , такие как оператор **map_to** используются для управления выполнением таких операторов HDL , как *, + и -.

Директивы запуска и останова трансляции

Директивы трансляции запускают и останавливают трансляцию исходного VHDL файла с помощью FPGA Express.

-- pragma translate_off
-- pragma translate_on

Директивы **translate_off** и **translate_on** заставляют FPGA Express запускать и останавливать синтез исходного кода VHDL . Тем не менее , код VHDL между этими двумя директивами проверяется на предмет синтаксических ошибок. Трансляция разрешена в начале каждого исходного файла VHDL. Вы можете использовать директивы **translate_off** и **translate_on** в любом месте текста.

Директивы **synthesis_off** и **synthesis_on** представляют собой механизм , рекомендуемый для укрывания конструкций , предназначенных только для моделирования , от синтеза. Любой тест между этими директивами проверяется на предмет синтаксиса , но никакого аппаратного обеспечения для него не синтезируется. Поведение директив **synthesis_off** и **synthesis_on** не зависит от переменной **hdlin_translate_off_skip_text** .

В примере 9-1 показано , как вы можете использовать директивы для защиты драйвера моделирования.

Пример 9-1. Использование директив synthesis_on и synthesis_off .

-- Следующий тестовый драйвер для объекта EXAMPLE

```
-- не должен транслироваться :
--
-- pragma synthesis_off
-- Трансляция остановлена

entity DRIVER is
end;

architecture VHDL of DRIVER is
        signal A, B : INTEGER range 0 to 255;
        signal SUM : INTEGER range 0 to 511;

        component EXAMPLE
                port (  A, B: in INTEGER range 0 to 255;
                        SUM: out INTEGER range 0 to 511);
        end component;
begin

        U1: EXAMPLE port map(A, B, SUM);
        process
        begin
                for I in 0 to 255 loop
                        for J in 0 to 255 loop
                                A <= I;
                                B <= J;
                                wait for 10 ns;
                                assert SUM = A + B;
                        end loop;
                end loop;
        end process;
end;

-- pragma synthesis_on
-- Начиная отсюда , код транслируется

entity EXAMPLE is
        port (  A, B: in INTEGER range 0 to 255;
                SUM: out INTEGER range 0 to 511);
end;

architecture VHDL of EXAMPLE is
begin
        SUM <= A + B;
end;
```

Директивы функции разрешения

Директивы функции разрешения определяют функцию разрешения , соответствующую разрешаемым сигналам (см. «Объявления сигналов» в Главе 3). FPGA Express в настоящее время не поддерживает произвольные функции разрешения. Он поддерживает следующие три метода :

```
-- pragma resolution_method wired_and
-- pragma resolution_method wired_or
-- pragma resolution_method three_state
```

Примечание : Не соединяйте сигналы , которые используют различные функции разрешения. FPGA Express поддерживает только одну функцию разрешения на цепь.

Директивы импликации компонента размещают подпрограммы VHDL в существующие компоненты или объекты VHDL. Эти директивы описаны в разделе «Размещение подпрограмм в компоненты» в Главе 6 :

-- pragma map_to_entity *entity_name*

-- pragma return_port_name *port_name*

Глава 10. Блоки объявлений Synopsys

В данную реализацию включены три блока объявлений Synopsys :

- *std_logic_1164*
  Определяет стандарт для разработчиков , используемый при описании взаимосвязи типов данных , применяемых в моделировании VHDL.
- *std_logic_arith*
  Обеспечивает набор арифметических функций , функций преобразования и сравнения для типов SIGNED , UNSIGNED , INTEGER , STD_ULOGIC , STD_LOGIC и STD_LOGIC_VECTOR.
- *std_logic_misc*
  Определяет поддерживаемые типы , подтипы , константы и функции для блока объявлений std_logic_1164.

Блок объявлений std_logic_1164

Этот блок определяет IEEE стандарт для разработчиков , используемый при описании взаимосвязи типов данных , применяемых в моделировании VHDL. Логическая система , определенная в этом блоке , может оказаться недостаточной для моделирования переключаемых транзисторов , поскольку такое требование находится за пределами ее возможностей. Кроме того , математика , примитивы и временные стандарты рассматривают ортогональные проблемы , как будто они относятся к этому блоку , и , следовательно , находятся за пределами его компетенции.

Блок объявлений **std_logic_1164** содержит директивы синтеза Synopsys . Три функции , однако , в настоящее время не поддерживаются для синтеза : **rising_edge** , **falling_edge** и **is_x** . Для использования этого блока в исходном файле VHDL включите следующие строки в его начало :

library IEEE;

use IEEE.std_logic_1164.all;

Когда вы анализируете исходный файл VHDL , FPGA Express автоматически находит библиотеку IEEE и блок **std_logic_1164**. Однако , вы должны анализировать блоки **use** , не содержащиеся в библиотеках IEEE и Synopsys , до обработки исходного файла , который их использует.

Блок объявлений std_logic_arith

Функции , определенные в блоке **std_logic_arith** , обеспечивают преобразование в и из предопределенного VHDL типа данных **INTEGER** , а также арифметические , сравнительные и Булевские операции. Этот блок позволяет вам выполнять арифметические операции и численные сравнения над данными типа массив. Блок определяет некоторые арифметические операторы (+, -, *, и **abs**) и операторы сравнения (<, >, <=, >=, =, и /=). Заметим , что IEEE VHDL не определяет арифметические операторы для массивов , а операторы сравнения определяет в такой интерпретации, которая не согласуется с арифметической интерпретацией значений массивов.

Этот блок также определяет два основных типа данных для своего собственного использования : **UNSIGNED** и **SIGNED** . Детали можно найти в разделе «Типы данных Synopsys» позже в этом приложении. Блок объявлений **std_logic_arith** является полностью допустимым для VHDL ; вы можете использовать его как для синтеза , так и для моделирования. Этот блок может быть сконфигурирован для работы с любыми массивами однобитовых типов. Вы кодируете однобитовые типы с помощью атрибута **ENUM_ENCODING**.

Вы можете сделать векторный тип (например , **std_logic_vector**) синонимом **SIGNED** или **UNSIGNED** . При этом , если вы планируете использовать в основном числа **UNSIGNED** , то вам не нужно будет преобразовывать ваш векторный тип , чтобы вызвать функции **UNSIGNED** . Недостатком такого синонимирования является то , что придется переопределять стандартные функции сравнения VHDL (=, /=, <, >, <=, и >=).

В таблице 9-1 показано , как стандартные функции сравнения для **BIT_VECTOR** не согласуются с функциями **SIGNED** и **UNSIGNED**.

Таблица 9-1. Функции сравнения UNSIGNED , SIGNED и BIT_VECTOR .

ARG1	op	ARG2	UNSIGNED	SIGNED	BIT_VECTOR
"000"	=	"000"	TRUE	TRUE	TRUE
"00"	=	"000"	TRUE	TRUE	FALSE
"100"	=	"0100"	TRUE	FALSE	FALSE
"000"	<	"000"	FALSE	FALSE	FALSE
"00"	<	"000"	FALSE	FALSE	TRUE
"100"	<	"0100"	FALSE	TRUE	FALSE

Использование блока объявлений

Блок объявлений **std_logic_arith** находится в директории **$synopsys/packages/IEEE/src/ std_logic_arith.vhd** корневого каталога Synopsys. Для использования этого блока в исходном файле VHDL включите следующие строки в его начало :

```
library IEEE;
use IEEE.std_logic_arith.all;
```

Блоки объявлений Synopsys предварительно проанализированы и не требуют дополнительного анализа.

Модификация блока объявлений

Блок объявлений **std_logic_arith** записан на стандартном языке VHDL. Вы можете модифицировать его или добавить что-то свое. Затем будет синтезировано соответствующее аппаратное обеспечение. Например , для преобразования вектора многозначительной логики в **INTEGER** вы можете написать функцию , показанную в примере 9-1. Эта функция **MVL_TO_INTEGER** возвращает целое значение , соответствующее вектору , когда вектор интерпретируется как беззнаковое (натуральное) число. Если в векторе встречаются неизвестные значения , то возвращается -1.

Пример 9-1. Новая функция на базе блока объявлений std_logic_arith .

```
library IEEE;
use IEEE.std_logic_1164.all;

function MVL_TO_INTEGER(ARG : MVL_VECTOR)
        return INTEGER is
        -- pragma built_in SYN_FEED_THRU
        variable uns: UNSIGNED (ARG'range);
begin
        for i in ARG'range loop
                case ARG(i) is
                        when '0' | 'L' => uns(i) := '0';
                        when '1' | 'H' => uns(i) := '1';
                        when others => return -1;
                end case;
        end loop;
        return CONV_INTEGER(uns);
end;
```

Отметим использование функции **CONV_INTEGER** в примере 9-1. FPGA Express выполняет почти весь синтез прямо из описаний VHDL. Однако , некоторые функции соединены аппаратно для большей эффективности. Эти функции могут быть идентифицированы с помощью следующего комментария в их объявлении :

```
-- pragma built_in
```

Этот оператор помечает функцию как специальную , заставляя игнорировать ее тело. Модификация тела *не* изменяет синтезируемую логику до тех пор , пока вы не уберете комментарий **built_in**. Если вы хотите достичь новой функциональности , используйте функции **built_in** ; это более эффективный способ , чем удаление **built_in** и модификация тела.

Типы данных

Блок объявлений **std_logic_arith** определяет два типа данных - **UNSIGNED** и **SIGNED :**

```
type UNSIGNED is array (natural range <>) of std_logic;
type SIGNED is array (natural range <>) of std_logic;
```

Эти типы похожи на предопределенный VHDL тип **BIT_VECTOR** , но блок **std_logic_arith** определяет интерпретацию переменных и сигналов этих типов как численные значения. С помощью записи преобразования **install_vhdl** вы можете изменить эти типы данных для массивов других однобитовых типов.

UNSIGNED

Тип данных **UNSIGNED** представляет беззнаковое численное значение. FPGA Express интерпретирует число в виде двоичного представления , у которого самый левый бит является старшим значащим. Например , десятичное число 8 может быть представлено как

UNSIGNED'("1000")

Когда вы объявляете переменные или сигналы типа **UNSIGNED** , то наибольший вектор хранит наибольшее число. Четырехбитовая переменная хранит значения до десятичного 15 ; восьмибитовая - до 255 и т.д. По определению , отрицательные числа не могут быть представлены в переменных типа **UNSIGNED**. Ноль является наименьшим числом , которое может быть представлено. В примере 9-2 проиллюстрированы некоторые объявления **UNSIGNED**. Заметим , что старший значащий бит является самым левым в границе массива.

Пример 9-2. Объявления UNSIGNED .

```
variable VAR: UNSIGNED (1 to 10);
-- 11-битовое число
-- VAR(VAR'left) = VAR(1) является старшим значащим битом

signal SIG: UNSIGNED (5 downto 0);
-- 6-битовое число
-- SIG(SIG'left) = SIG(5) является старшим значащим битом
```

SIGNED

Тип данных **SIGNED** представляет знаковое численное значение. FPGA Express интерпретирует число в двоичном дополнительном формате со знаковым битом , который стоит слева. Например , вы можете представить десятичное 5 и -5 как

```
SIGNED'("0101")      -- представляет +5
SIGNED'("1011")      -- представляет -5
```

Когда вы объявляете переменные или сигналы типа **SIGNED ,** то наибольший вектор хранит наибольшее число. Четырехбитовая переменная хранит значение от -8 до 7 ; восьмибитовая - от –128 до 127. Заметим , что величина типа **SIGNED** не может хранить значение такой же величины , как **UNSIGNED** , при одинаковой битовой ширине. В примере 9-3 показаны некоторые объявления **SIGNED**. Заметим , что знаковый бит является самым левым.

Пример 9-3. Объявления SIGNED .

```
variable S_VAR: SIGNED (1 to 10);
-- 11-битовое число
-- S_VAR(S_VAR'left) = S_VAR(1) знаковый бит

signal S_SIG: SIGNED (5 downto 0);
-- 6-битовое число
-- S_SIG(S_SIG'left) = S_SIG(5) знаковый бит
```

Блок объявлений **std_logic_arith** обеспечивает три набора функций для преобразования значений между типами **UNSIGNED** и **SIGNED** , а также предопределенный тип **INTEGER** . Этот блок также обеспечивает **std_logic_vector** . В примере 9-4 показаны объявления этих функций преобразования . Показаны типы **BIT** и **BIT_VECTOR**.

Пример 9-4. Функции преобразования.

```
subtype SMALL_INT is INTEGER range 0 to 1;
function CONV_INTEGER(ARG: INTEGER) return INTEGER;
function CONV_INTEGER(ARG: UNSIGNED) return INTEGER;
function CONV_INTEGER(ARG: SIGNED) return INTEGER;
function CONV_INTEGER(ARG: STD_ULOGIC) return SMALL_INT;
function CONV_UNSIGNED(   ARG: INTEGER;
                          SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(   ARG: UNSIGNED;
                          SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(   ARG: SIGNED;
                          SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(   ARG: STD_ULOGIC;
                          SIZE: INTEGER) return UNSIGNED;
function CONV_SIGNED(     ARG: INTEGER;
                          SIZE: INTEGER) return SIGNED;
function CONV_SIGNED(     ARG: UNSIGNED;
                          SIZE: INTEGER) return SIGNED;
function CONV_SIGNED(     ARG: SIGNED;
                          SIZE: INTEGER) return SIGNED;
function CONV_SIGNED(     ARG: STD_ULOGIC;
                          SIZE: INTEGER) return SIGNED;
function CONV_STD_LOGIC_VECTOR(ARG: INTEGER;
                                SIZE: INTEGER) return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: UNSIGNED;
                                SIZE: INTEGER) return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: SIGNED;
                                SIZE: INTEGER) return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: STD_ULOGIC;
                                SIZE: INTEGER) return STD_LOGIC_VECTOR;
```

Заметим , что существует четыре версии каждой функции преобразования. Механизм перегрузки операторов VHDL определяет правильную версию по типу аргумента в вызове функции.

Функции **CONV_INTEGER** преобразуют аргумент типа **INTEGER** , **UNSIGNED** , **SIGNED** или **STD_ULOGIC** в **INTEGER**. Функции **CONV_UNSIGNED** и **CONV_SIGNED** преобразуют аргумент типа **INTEGER** , **UNSIGNED** , **SIGNED** или **STD_ULOGIC** в **UNSIGNED** или **SIGNED** с битовой шириной **SIZE** . Функции **CONV_INTEGER** ограничивают размер своих операндов. VHDL определяет значения **INTEGER** в диапазоне от -2147483647 до 2147483647. Этот диапазон соответствует 31-битовому значению **UNSIGNED** или 32-битовому значению **SIGNED**. Вы не можете преобразовать аргумент , находящийся за пределами этого диапазона , в **INTEGER .**

Функции **CONV_UNSIGNED** и **CONV_SIGNED** требуют два операнда. Первый операнд является преобразуемым значением. Второй - типа **INTEGER** - устанавливает ожидаемый размер преобразованного результата. Например , следующий вызов функции возвращает 10-битовое значение **UNSIGNED** , представляющее величину **sig**.

```
ten_unsigned_bits := CONV_UNSIGNED(sig, 10);
```

Если значение , пропускаемое через **CONV_UNSIGNED** или **CONV_SIGNED** , меньше , чем ожидаемая битовая ширина , то значение соответственно расширяется битами. FPGA Express помещает нули в старшие значащие (левые) биты для возвращаемого значения **UNSIGNED** и использует знаковое расширение для возвращаемого значения **SIGNED**. Вы можете использовать функции пре-

образования для расширения количества бит в числе , даже если преобразование не требуется. Например :

CONV_SIGNED(SIGNED'("110"), 8) % "11111110"

UNSIGNED или **SIGNED** возвращаемое значение усекается , если битовая ширина слишком мала для хранения аргумента **ARG**. Например :

CONV_SIGNED(UNSIGNED'("1101010"), 3) % "010"

Арифметические функции

Блок объявлений **std_logic_arith** обеспечивает арифметические функции для работы вместе с типами данных Synopsys **UNSIGNED** и **SIGNED** и предопределенными типами **STD_ULOGIC** и **INTEGER** . Эти функции производят сумматоры и вычитатели.

Существует два набора арифметических функций : бинарные с двумя аргументами (такие как **A+B** или **A*B**) и унарные с одним аргументом (такие как **-A**). Объявления для этих функций показаны в примере 9-5 и 9-6.

Пример 9-5. Бинарные арифметические функции.

```
function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED; R: SIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: SIGNED) return SIGNED;
function "+"(L: SIGNED; R: UNSIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
function "+"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED; R: INTEGER) return SIGNED;
function "+"(L: INTEGER; R: SIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
function "+"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
function "+"(L: STD_ULOGIC; R: SIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "+"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "+"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "+"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "+"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED; R: SIGNED) return SIGNED;
function "-"(L: UNSIGNED; R: SIGNED) return SIGNED;
function "-"(L: SIGNED; R: UNSIGNED) return SIGNED;
function "-"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
function "-"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED; R: INTEGER) return SIGNED;
function "-"(L: INTEGER; R: SIGNED) return SIGNED;
function "-"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
function "-"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
function "-"(L: STD_ULOGIC; R: SIGNED) return SIGNED;
function "-"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
```

```
function "-"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "-"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "-"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "-"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "-"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;
function "*"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "*"(L: SIGNED; R: SIGNED) return SIGNED;
function "*"(L: SIGNED; R: UNSIGNED) return SIGNED;
function "*"(L: UNSIGNED; R: SIGNED) return SIGNED;
```

Пример 9-6. Унарные арифметические функции.

```
function "+"(L: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED) return SIGNED;
function "-"(L: SIGNED) return SIGNED;
function "ABS"(L: SIGNED) return SIGNED;
```

Эти функции определяют ширину своих возвращаемых значений следующим образом :
1. Когда присутствует только аргумент **UNSIGNED** или **SIGNED** , то ширина возвращаемого значения является такой же , как у этого аргумента.
2. Когда оба аргумента либо **UNSIGNED** , либо **SIGNED** , то ширина возвращаемого значения равна наибольшей ширине аргумента. Исключение составляет тот случай , когда число **UNSIGNED** складывается с или вычитается из числа **SIGNED** такого же или меньшего размера ; тогда возвращаемое значение является числом **SIGNED** , которое на один бит шире , чем аргумент **UNSIGNED**. Такой размер гарантирует , что возвращаемое значение является достаточно большим , чтобы хранить любое (положительное) значение аргумента **UNSIGNED**.

Количество бит , возвращаемых операторами + и - , проиллюстрировано в табл.9-2.

```
signal U4: UNSIGNED (3 downto 0);
signal U8: UNSIGNED (7 downto 0);
signal S4: SIGNED (3 downto 0);
signal S8: SIGNED (7 downto 0);
```

Таблица 9-2. Количество бит , возвращаемых операциями + и - .

+ или -	U4	U8	S4	S8
U4	4	8	5	8
U8	8	8	9	9
S4	5	9	4	8
S8	8	9	8	8

В некоторых случаях вам может потребоваться бит переноса для операций + или - . Чтобы осуществить это , расширьте больший операнд на один бит. Старший бит возвращаемого значения будет являться битом переноса , как показано в примере 9-7.

Пример 9-7. Использование бита переноса.

```
process
        variable a, b, sum: UNSIGNED (7 downto 0);
        variable temp: UNSIGNED (8 downto 0);
        variable carry: BIT;
```

```
begin
        temp := CONV_UNSIGNED(a,9) + b;
        sum := temp(7 downto 0);
        carry := temp(8);
end process;
```

Функции сравнения

Блок объявлений **std_logic_arith** обеспечивает функции для сравнения значений типов данных **UNSIGNED** и **SIGNED** друг с другом , а также с предопределенным типом **INTEGER** . FPGA Express сравнивает *численные* значения аргументов , возвращая Булевское значение. Например, следующее выражение вычисляет **TRUE** .

```
UNSIGNED'("001") > SIGNED'("111")
```

Функции сравнения **std_logic_arith** похожи на встроенные функции сравнения VHDL. Единственное различие заключается в том , что функции **std_logic_arith** приспособлены к знаковым числам и различной битовой ширине. Предопределенные функции сравнения VHDL выполняют битовое сравнение и , таким образом , не имеют корректной семантики для сравнения численных значений (см. «Операторы сравнения» в Главе 5).

Эти функции производят компараторы. Объявления функций перечислены в двух группах : функции упорядочения (<, <=, > и >=) и функции равенства (= и /=) в примерах 9-8 и 9-9.

Пример 9-8. Функции упорядочения.

```
function "<"(L: UNSIGNED; R: UNSIGNED) return Boolean;
function "<"(L: SIGNED; R: SIGNED) return Boolean;
function "<"(L: UNSIGNED; R: SIGNED) return Boolean;
function "<"(L: SIGNED; R: UNSIGNED) return Boolean;
function "<"(L: UNSIGNED; R: INTEGER) return Boolean;
function "<"(L: INTEGER; R: UNSIGNED) return Boolean;
function "<"(L: SIGNED; R: INTEGER) return Boolean;
function "<"(L: INTEGER; R: SIGNED) return Boolean;
function "<="(L: UNSIGNED; R: UNSIGNED) return Boolean;
function "<="(L: SIGNED; R: SIGNED) return Boolean;
function "<="(L: UNSIGNED; R: SIGNED) return Boolean;
function "<="(L: SIGNED; R: UNSIGNED) return Boolean;
function "<="(L: UNSIGNED; R: INTEGER) return Boolean;
function "<="(L: INTEGER; R: UNSIGNED) return Boolean;
function "<="(L: SIGNED; R: INTEGER) return Boolean;
function "<="(L: INTEGER; R: SIGNED) return Boolean;
function "" functions">">"(L: UNSIGNED; R: UNSIGNED) return Boolean;
function ">"(L: SIGNED; R: SIGNED) return Boolean;
function ">"(L: UNSIGNED; R: SIGNED) return Boolean;
function ">"(L: SIGNED; R: UNSIGNED) return Boolean;
function ">"(L: UNSIGNED; R: INTEGER) return Boolean;
function ">"(L: INTEGER; R: UNSIGNED) return Boolean;
function ">"(L: SIGNED; R: INTEGER) return Boolean;
function ">"(L: INTEGER; R: SIGNED) return Boolean;
function ="" functions">">="(L: UNSIGNED; R: UNSIGNED) return Boolean;
function ">="(L: SIGNED; R: SIGNED) return Boolean;
function ">="(L: UNSIGNED; R: SIGNED) return Boolean;
function ">="(L: SIGNED; R: UNSIGNED) return Boolean;
function ">="(L: UNSIGNED; R: INTEGER) return Boolean;
function ">="(L: INTEGER; R: UNSIGNED) return Boolean;
function ">="(L: SIGNED; R: INTEGER) return Boolean;
function ">="(L: INTEGER; R: SIGNED) return Boolean;
```

Пример 9-9. Функции равенства.

```
function "="(L: UNSIGNED; R: UNSIGNED) return Boolean;
function "="(L: SIGNED; R: SIGNED) return Boolean;
function "="(L: UNSIGNED; R: SIGNED) return Boolean;
function "="(L: SIGNED; R: UNSIGNED) return Boolean;
function "="(L: UNSIGNED; R: INTEGER) return Boolean;
function "="(L: INTEGER; R: UNSIGNED) return Boolean;
function "="(L: SIGNED; R: INTEGER) return Boolean;
function "="(L: INTEGER; R: SIGNED) return Boolean;
function "/="(L: UNSIGNED; R: UNSIGNED) return Boolean;
function "/="(L: SIGNED; R: SIGNED) return Boolean;
function "/="(L: UNSIGNED; R: SIGNED) return Boolean;
function "/="(L: SIGNED; R: UNSIGNED) return Boolean;
function "/="(L: UNSIGNED; R: INTEGER) return Boolean;
function "/="(L: INTEGER; R: UNSIGNED) return Boolean;
function "/="(L: SIGNED; R: INTEGER) return Boolean;
function "/="(L: INTEGER; R: SIGNED) return Boolean;
```

Функции сдвига

Блок объявлений **std_logic_arith** обеспечивает функции для сдвига бит в числах типов **SIGNED** и **UNSIGNED**. Эти функции производят устройства сдвига. В примере 9-10 приведены объявления функций сдвига.

Пример 9-10. Функции сдвига.

```
function SHL(  ARG: UNSIGNED;
               COUNT: UNSIGNED) return UNSIGNED;
function SHL(  ARG: SIGNED;
               COUNT: UNSIGNED) return SIGNED;
function SHR(  ARG: UNSIGNED;
               COUNT: UNSIGNED) return UNSIGNED;
function SHR(  ARG: SIGNED;
               COUNT: UNSIGNED) return SIGNED;
```

Функция **SHL** сдвигает биты аргумента **ARG** *влево* на **COUNT** бит. **SHR** сдвигает биты аргумента **ARG** *вправо* на **COUNT** бит. Функции **SHL** работают одинаково как для **UNSIGNED** , так и для **SIGNED** значений **ARG** , осуществляя , по необходимости , нулевой сдвиг. Функции **SHR** трактуют **UNSIGNED** и **SIGNED** значения по-разному. Если **ARG** является числом типа **UNSIGNED** , освобождающиеся биты заполняются нулями ; если же **ARG** является числом типа **SIGNED** , то освобождающиеся биты копируются из знакового бита **ARG**. В примере 9-11 показаны некоторые вызовы функций сдвига и возвращаемые ими значения.

Пример 9-11. Операции сдвига.

```
variable U1, U2: UNSIGNED (7 downto 0);
variable S1, S2: SIGNED (7 downto 0);
variable COUNT: UNSIGNED (1 downto 0);
. . .
U1 := "01101011";
U2 := "11101011";

S1 := "01101011";
S2 := "11101011";

COUNT := CONV_UNSIGNED(ARG => 3, SIZE => 2);
```

```
. . .
SHL(U1, COUNT) = "01011000"
SHL(S1, COUNT) = "01011000"
SHL(U2, COUNT) = "01011000"
SHL(S2, COUNT) = "01011000"

SHR(U1, COUNT) = "00001101"
SHR(S1, COUNT) = "00001101"
SHR(U2, COUNT) = "00011101"
SHR(S2, COUNT) = "11111101"
```

Умножение с помощью сдвига

Вы можете использовать операции сдвига для простого умножения и деления беззнаковых чисел , если вы умножаете или делите на степень двойки. Например , для деления следующей беззнаковой переменной **U** на 4 :

```
variable U: UNSIGNED (7 downto 0) := "11010101";
variable quarter_U: UNSIGNED (5 downto 0);

quarter_U := SHR(U, "01");
```

Атрибут ENUM_ENCODING

Размещайте атрибут синтеза **ENUM_ENCODING** в вашем основном логическом типе (см. «Кодирование перечисления» в Главе 4). Этот атрибут позволяет FPGA Express корректно интерпретировать вашу логику.

pragma built_in

Помечайте ваши основные логические функции псевдокомментарием **built_in**. Он позволяет FPGA Express более легко интерпретировать ваши логические функции. Когда вы используете комментарий **built_in** , FPGA Express анализирует , но игнорирует тело функции. Вместо этого FPGA Express непосредственно заменяет функцию соответствующей логикой. Вы можете не использовать комментарии **built_in** ; однако результатом их применения является ускорение работы примерно в десять раз.

Используйте псевдокомментарии **built_in** путем помещения комментария в части объявления функции. FPGA Express интерпретирует комментарий как директиву , если первым его словом является **pragma** . В примере 9-12 показано использование **built_in**.

Пример 9-12. Использование псевдокомментария built_in .

```
function "XOR" (L, R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
        -- pragma built_in SYN_XOR
begin
        if (L = '1') xor (R = '1') then
                return '1';
        else
                return '0';
        end if;
end "XOR";
```

Логические функции с двумя аргументами

Synopsys обеспечивает шесть встроенных функций для выполнения логических операций с двумя аргументами :

- **SYN_AND**

- **SYN_OR**
- **SYN_NAND**
- **SYN_NOR**
- **SYN_XOR**
- **SYN_XNOR**

Вы можете использовать эти функции с однобитовыми аргументами или равновеликими массивами одиночных бит. В примере 9-13 показана функция , которая генерирует логическое И двух равновеликих массивов.

Пример 9-13. Встроенное И для массивов.

```
function "AND" (L, R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
        -- pragma built_in SYN_AND

        variable MY_L: STD_LOGIC_VECTOR (L'length-1 downto 0);
        variable MY_R: STD_LOGIC_VECTOR (L'length-1 downto 0);
        variable RESULT: STD_LOGIC_VECTOR (L'length-1 downto 0);

begin

        assert L'length = R'length;
        MY_L := L;
        MY_R := R;
        for i in RESULT'range loop
                if (MY_L(i) = '1') and (MY_R(i) = '1') then
                        RESULT(i) := '1';
                else
                        RESULT(i) := '0';
                end if;
        end loop;
        return RESULT;

end "AND";
```

Логические функции с одним аргументом

Synopsys обеспечивает две встроенные функции для выполнения логической операции с одним аргументом :

- **SYN_NOT**
- **SYN_BUF**

Вы можете использовать эти функции с однобитовыми аргументами или равновеликими массивами одиночных бит. В примере 9-14 показана функция , которая генерирует логическое НЕ массива.

Пример 9-14. Встроенное НЕ для массивов.

```
function "NOT" (L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
        -- pragma built_in SYN_NOT
        variable MY_L: STD_LOGIC_VECTOR (L'length-1 downto 0);
        variable RESULT: STD_LOGIC_VECTOR (L'length-1 downto 0);
begin
        MY_L := L;
        for i in result'range loop
                if (MY_L(i) = '0' or MY_L(i) = 'L') then
                                RESULT(i) := '1';
                        elsif (MY_L(i) = '1' or MY_L(i) = 'H') then
                                RESULT(i) := '0';
```

```
                    else
                        RESULT(i) := 'X';
                end if;
        end loop;
        return RESULT;
end "NOT";
```

Преобразование типа

Встроенная функция **SYN_FEED_THRU** выполняет быстрое преобразование типа между не связанными типами. Логика , синтезируемая из **SYN_FEED_THRU** , соединяет одиночный вход функции с возвращаемым значением. Такая связь может сэкономить время процессора , требуемое для процесса , выполняющего функцию преобразования , как показано в примере 9-15.

Пример 9-15. Использование SYN_FEED_THRU.

```
type COLOR is (RED, GREEN, BLUE);
attribute ENUM_ENCODING : STRING;
attribute ENUM_ENCODING of COLOR : type is "01 10 11";
...
function COLOR_TO_BV (L: COLOR) return BIT_VECTOR is
        -- pragma built_in SYN_FEED_THRU
begin
        case L is
                when RED => return "01";
                when GREEN => return "10";
                when BLUE => return "11";
        end case;
end COLOR_TO_BV;
```

Директива translate_off

Если в блоке объявлений ваших типов существуют конструкции , не поддерживаемые для синтеза или приводящие к генерации предупреждающих сообщений , то вам может потребоваться директива FPGA Express **–synopsys translate_off** . Вы можете сделать свободным использование директивы **translate_off** , если вы задействовали псевдокомментарий **built_in** , поскольку FPGA Express игнорирует тела функций **built_in**. Примеры , иллюстрирующие применение директивы **translate_off**, можно увидеть в блоке объявлений **std_logic_arith.vhd**.

Блок объявлений std_logic_misc

Блок объявлений **std_logic_misc** расположен в директории библиотек Synopsys (**$synopsys/packages/IEEE/src/std_logic_misc.vhd**). Этот блок объявляет основные типы данных , поддерживаемые семейством Synopsys VSS Family.

Булевские функции редукции используют один аргумент , массивы бит , и возвращают одиночный бит. Например , И-редукция **"101"** равна **"0"**, логическому И всех трех бит. Различные функции в блоке объявлений **std_logic_misc** обеспечивают Булевские операции редукции для предопределенного типа **STD_LOGIC_VECTOR** . В примере 9-16 показаны объявления этих функций.

Пример 9-16. Булевские функции редукции.

```
function AND_REDUCE (ARG: STD_LOGIC_VECTOR) return UX01;
function NAND_REDUCE (ARG: STD_LOGIC_VECTOR) return UX01;
function OR_REDUCE (ARG: STD_LOGIC_VECTOR) return UX01;
function NOR_REDUCE (ARG: STD_LOGIC_VECTOR) return UX01;
```

```
function XOR_REDUCE (ARG: STD_LOGIC_VECTOR) return UX01;
function XNOR_REDUCE (ARG: STD_LOGIC_VECTOR) return UX01;
function AND_REDUCE (ARG: STD_ULOGIC_VECTOR) return UX01;
function NAND_REDUCE (ARG: STD_ULOGIC_VECTOR) return UX01;
function OR_REDUCE (ARG: STD_ULOGIC_VECTOR) return UX01;
function NOR_REDUCE (ARG: STD_ULOGIC_VECTOR) return UX01;
function XOR_REDUCE (ARG: STD_ULOGIC_VECTOR) return UX01;
function XNOR_REDUCE (ARG: STD_ULOGIC_VECTOR) return UX01;
```

Эти функции комбинируют биты **STD_LOGIC_VECTOR** , как показывают их имена. Например, **XOR_REDUCE** возвращает исключающее ИЛИ всех бит **ARG**. В примере 9-17 показаны некоторые вызовы функций редукции и возвращаемые ими значения.

Пример 9-17. Булевские операции редукции.

```
AND_REDUCE("111") = '1'
AND_REDUCE("011") = '0'
OR_REDUCE("000") = '0'
OR_REDUCE("001") = '1'
XOR_REDUCE("100") = '1'
XOR_REDUCE("101") = '0'
NAND_REDUCE("111") = '0'
NAND_REDUCE("011") = '1'
NOR_REDUCE("000") = '1'
NOR_REDUCE("001") = '0'
XNOR_REDUCE("100") = '0'
XNOR_REDUCE("101") = '1'
```

Глава 11. Конструкции HDL

Многие языковые конструкции VHDL не имеют отношения к синтезу , хотя и пригодны для моделирования и других стадий проектирования. Поскольку эти конструкции не могут быть синтезированы , они не поддерживаются FPGA Express. В этом приложении приведен список всех языковых конструкций VHDL с уровнем поддержки для каждой из них , а также список зарезервированных слов VHDL.

Данное приложение описывает :

- поддерживаемые конструкции VHDL
- зарезервированные слова VHDL

Поддерживаемые конструкции VHDL

Конструкция может полностью поддерживаться , игнорироваться или не поддерживаться совсем. Игнорируемые и не поддерживаемые конструкции определяются следующим образом :

- Игнорируемыми называются конструкции , которые разрешены в исходных файлах VHDL , но игнорируются FPGA Express.
- Не поддерживаемыми называются конструкции , которые не разрешены в исходных файлах VHDL и которые помечаются FPGA Express как ошибочные. Если ошибка обнаруживается в описании VHDL , то оно не транслируется (не синтезируется).

  Конструкции перечислены в следующем порядке :
  - проектные единицы
  - типы данных
  - объявления
  - спецификации
  - имена
  - операторы

- операнды и выражения
- последовательные операторы
- параллельные операторы
- предопределенная среда языка

Проектные единицы

объект

Операторная часть объекта игнорируется. Поддерживаются основные параметры (Generics) , но только типа **INTEGER** . Значения по умолчанию для портов игнорируются.

архитектура

Разрешены множественные архитектуры. Не поддерживается глобальная сигнальная взаимосвязь между архитектурами.

конфигурация

Поддерживаются объявления и блоки конфигураций , но только для определенной архитектуры высшего уровня и объекта высшего уровня. Атрибутные спецификации . предложения **use** , компонентные конфигурации и вложенные блоки конфигураций не поддерживаются.

блок объявлений

Блоки объявлений поддерживаются полностью.

библиотека

Поддерживаются библиотеки и раздельная компиляция.

подпрограмма

Не поддерживаются значения по умолчанию для параметров. Не поддерживается назначение индексным и скользящим именам неограниченных параметров **out** , пока действительный параметр является идентификатором.

Не поддерживается рекурсия подпрограммы , если она не ограничена статическим значением.

Поддерживаются функции разрешения только типов соединенной логики и третьего состояния.

Подпрограммы могут быть объявлены только в блоках объявлений и в объявительной части архитектуры.

Типы данных

перечисляемый

Перечисление полностью поддерживается.

целый

Не поддерживается бесконечно точная арифметика. Целые типы автоматически преобразуются в битовые вектора , ширина которых мала настолько , чтобы разместить все возможные значения из диапазона данного типа ; либо в беззнаковые двоичные числа для неотрицательных диапазонов , либо в двоичный дополнительный формат для диапазонов , включающих отрицательные числа.

физический

Объявления физического типа игнорируются. Использование физических типов игнорируется в спецификациях задержки.

с плавающей точкой

Объявления типа с плавающей точкой игнорируются. Использование вещественных типов не поддерживается , за исключением констант , используемых с атрибутами , определенными Synopsys (см. Главу 9).

массив

Не поддерживаются индексы и диапазоны массивов , отличные от целых. Не поддерживаются многоразмерные массивы , однако поддерживаются массивы массивов.

запись

Типы данных запись полностью поддерживаются.

доступ

Объявления типа доступ игнорируются , а использование типов доступ не поддерживается.

файл

Объявления типа файл игнорируются , а использование типов файл не поддерживается.

незавершенные объявления типов

Незавершенные объявления типов не поддерживаются.

Объявления

константа

Объявления констант поддерживаются за исключением задержанных объявлений.

сигнал

Объявления **register** и **bus** не поддерживаются. Функции разрешения поддерживаются только для функций объединения и третьего состояния. Объявления , отличные от глобального статического типа , не поддерживаются. Начальные значения не поддерживаются.

переменная

Объявления , отличные от глобального статического типа , не поддерживаются. Начальные значения не поддерживаются.

файл

Объявления файлов не поддерживаются.

интерфейс

buffer и **linkage** транслируются в **out** и **inout** , соответственно.

альтернативное имя (alias)

Объявления альтернативных имен не поддерживаются.

компонент

Компонентные объявления , которые перечисляют имена , отличные от допустимых имен объекта , не поддерживаются.

атрибут

Объявления атрибутов полностью поддерживаются. Однако , не поддерживается использование атрибутов , определенных пользователем.

Спецификации

атрибут

others и **all** не поддерживаются в спецификациях атрибутов. Пользовательские атрибуты могут быть определены , но их использование не поддерживается.

конфигурация

Спецификации конфигураций не поддерживаются.

разъединение

Спецификации разъединения не поддерживаются. Объявления атрибутов поддерживаются полностью. Однако использование атрибутов , определенных пользователем , не поддерживается.

Имена

простые

Простые имена поддерживаются полностью.

выборочные

Выборочные (*определенные*) имена за пределами предложения **use** не поддерживаются. Перезапись областей идентификаторов не поддерживается.

символы операторов

Символы операторов поддерживаются полностью.

индексные

Индексные имена поддерживаются за одним исключением. Не поддерживается индексирование неограниченного **out** параметра в процедуре.

скользящие

Скользящие имена поддерживаются за одним исключением. Использование скольжения неограниченного **out** параметра в процедуре не поддерживается , пока действительный параметр является идентификатором.

атрибут

Поддерживаются только следующие предопределенные атрибуты : **base** , **left** , **right** , **high** , **low**, **range** , **reverse_range** и **length.** Атрибуты **event** и **stable** поддерживаются только как описано в операторах **wait** и **if** (см. Главу 6).

Имена атрибутов , определенных пользователем , не поддерживаются.

Использование атрибутов с выборочными именами (**name.name'attribute**) не поддерживается.

Операторы

логические

Логические операторы поддерживаются полностью.

сравнительные

Операторы сравнения поддерживаются полностью.

сложения

Конкатенация и арифметические операторы поддерживаются полностью.

знаковые

Знаковые операторы поддерживаются полностью.

умножения

Оператор ***** (умножение) поддерживается полностью. Операторы **/** (деление) , **mod** и **rem** поддерживаются только когда оба операнда являются константами или когда правый операнд является константой - степенью двойки.

смешанные

Оператор ****** поддерживается только когда оба операнда являются константами или когда левый операнд является двойкой. Оператор **abs** поддерживается полностью.

перегрузка операторов

Перегрузка операторов поддерживается полностью.

операции короткого замыкания

Короткозамкнутое поведение операторов не поддерживается.

Операнды и выражения

базовые литералы

Базовые литералы поддерживаются полностью.

нулевые литералы

Нулевые скольжения , нулевые диапазоны и нулевые массивы не поддерживаются.

физические литералы

Физические литералы игнорируются.

строки

Строки поддерживаются полностью.

множества

Использование типов как множественных выборов не поддерживается. Записи-множества не поддерживаются.

вызовы функций

Преобразования функций на входных портах не поддерживаются , так как не поддерживаются преобразования типов на формальных портах в спецификации связи.

определенные выражения

Определенные выражения поддерживаются полностью.

преобразования типа

Преобразования типа поддерживаются полностью.

распределители (allocators)

Распределители не поддерживаются.

статические выражения

Статические выражения поддерживаются полностью.

универсальные выражения

Выражения с плавающей точкой не поддерживаются , исключая распознаваемого Synopsys определения атрибутов.

Выражения с бесконечной точностью не поддерживаются. Точность ограничена 32 битами ; все промежуточные результаты преобразуются в целые.

Последовательные операторы

wait (ожидание)

Оператор **wait** не поддерживается , пока не будет записан в одном из следующих видов :

wait until *clock* = *VALUE*;
wait until *clock*'event and *clock* = *VALUE*;
wait until not *clock*'stable and *clock* = *VALUE*;

где **VALUE** является **0** , **1** или перечисляемым литералом , который кодируется **0** или **1**. Оператор **wait** в таком виде интерпретируется как «ожидание падающего (**VALUE** равно **0**) или растущего (**VALUE** равно **1**) среза сигнала с именем **clock** .” Операторы **wait** не могут использоваться в подпрограммах или циклах **for**.

assertion (утверждение)

Операторы **assertion** игнорируются.

сигнал

Не поддерживается защищенное присваивание сигналу. **transport** и **after** игнорируются. Многочисленные сигнальные элементы в операторах присваивания не поддерживаются.

переменная

Операторы **variable** поддерживаются полностью.

вызов процедуры

Преобразование типа формальных параметров не поддерживается. Присваивание одиночным битам векторных портов не поддерживается.

if (условие)

Операторы **if** поддерживаются полностью.

case (выбор)

Операторы **case** поддерживаются полностью.

loop (цикл)

Циклы **for** поддерживаются с двумя ограничениями : диапазон индекса цикла должен быть глобально статическим , а тело цикла не должно содержать оператор **wait**. Циклы **while** поддерживаются , однако тело цикла должно содержать по меньшей мере один оператор **wait**. Операторы **loop** без схемы итерации (бесконечные циклы) поддерживаются , однако тело цикла должно содержать по меньшей мере один оператор **wait**.

next (следующий)

Операторы **next** поддерживаются полностью.

exit (выход)

Операторы **exit** поддерживаются полностью.

return (возврат)

Операторы **return** поддерживаются полностью.

null (нулевой)

Операторы **null** поддерживаются полностью.

Параллельные операторы

block

Защита операторов **block** не поддерживается. Порты и generics в операторах **block** не поддерживаются.

process

Списки чувствительности в операторах **process** игнорируются.

параллельный вызов процедуры

Операторы параллельного вызова процедур полностью поддерживаются.

параллельное утверждение

Операторы параллельного утверждения игнорируются.

параллельное присваивание сигналу

Ключевые слова **guarded** и **transport** игнорируются. Многочисленные сигналы не поддерживаются.

реализация компонента

Преобразование типа на формальном порту спецификации связи не поддерживается.

generate

Операторы **generate** поддерживаются полностью.

Предопределенная среда языка

mun severity_level

Тип **severity_level** не поддерживается.

mun time

Тип **time** не поддерживается.

функция now

Функция **now** не поддерживается.

блок объявлений TEXTIO

Блок объявлений **TEXTIO** не поддерживается.

предопределенные атрибуты

Предопределенные атрибуты не поддерживаются , за исключением **base** , **left** , **right** , **high** , **low**, **range** , **reverse_range** и **length** . Атрибуты **event** и **stable** поддерживаются только в операторах **if** и **wait** , как описано в Главе 6.

Зарезервированные слова VHDL

Следующие слова зарезервированы в языке VHDL и не могут использоваться в качестве идентификаторов :

abs ifselect
alias issubtype
architecture library to
attribute map units
body nand
nor wait
configuration with
disconnect open
else out
entity port
file
register
guarded return
access inseverity
all
array linkage transport
begin mod until
buffer new variable
case not when
constant of
downto or
elsif
exit procedure
for range
generate rem
after inout signal
and label then
assert loop type
block use
bus next
component null while
onxor
others
end package
process
function record
generic report