Продолжение. Начало в № 3`2009

Изучаем Active-HDL 7.1. Урок 10. Как проектировать символы

Александр ШАЛАГИНОВ
shalag@vt.cs.nstu.ru

Символ — это условное графическое изображение произвольного фрагмента цифровой схемы, представленное в виде черного ящика. Интерфейс (связь) символа с «внешним миром» осуществляется через его контакты (pins).

Символы иногда называют графическими моделями компонентов. Они нужны только при проектировании схем.

На схемах нижнего уровня иерархии символами обычно изображаются простые логические элементы и триггеры. На схемах более высоких уровней (например, **RTL** или **PMS**) целый узел, устройство или даже функциональный блок могут быть представлены как логический примитив, то есть символ.

Даже проект верхнего уровня иногда замещается символом. Такая необходимость возникает, например, при включении его в испытательный стенд **TestBench** (урок 3). Чтобы абстрагироваться от уровня сложности, любой объект, который представляется символом, принято называть компонентом.

На рис. 1 показаны символы из системной библиотеки **Spartan3**, представляющие компоненты различных уровней сложности: от простейших генераторов нуля **GND** и единицы **VCC** до 8-разрядного сумматора **ADD8**.

Функция, выполняемая символом, реализуется благодаря его внутреннему описанию, которое принято называть содержанием (представлением, реализацией) символа (английские названия **content**, **implementation**).

Внутреннее описание может быть изначально представлено в текстовом или графическом формате. В первом случае — это **HDL**-код или **EDIF** список цепей. Во втором случае — это схема более низкого уровня иерархии или диаграмма состояний цифрового автомата. Заметим, что схема и диаграмма позднее будут в любом случае конвертированы в текстовый формат (в **HDL**-описание или **EDIF** список цепей (только для схемы)).

Из сказанного становится понятным, почему внутреннее описание символа нередко называют функциональной (поведенческой) или структурной моделью.

Моделятору доступны только откомпилированные модели компонентов, которые хранятся в библиотеках в виде отдельных модулей. Специальная программа, называемая менеджером библиотек (**Library Manager**), управляет этими библиотеками, так что пользователю нет необходимости углубляться в организацию и файловую структуру физических библиотек.

Подводя итог, можно заключить, что любой полноценный компонент должен иметь два описания — внешнее (символ, графическую модель) и внутреннее (содержимое, реализацию, структурную или поведенческую модель). Оба описания связываются между собой одинаковым именем.

Конечно, вы можете создать символ без внутреннего содержания (**content**). Такой символ принято называть пустым символом (**Empty Symbol**). Пустые символы пригодны для рисования схем, но правильно смоделировать работу схемы, содержащей пустые символы, вам не удастся.

Если в качестве целевого проектирования выбран язык **VHDL**, то у вас появляется дополнительная и весьма полезная возможность — задать для одного символа несколько реализаций (внутренних описаний). Понятно, что только одна из них будет активной, то есть используемой при моделировании. По умолчанию это последняя откомпилированная архитектура.

В дальнейшем мы познакомимся с механизмом выбора активной реализации из списка возможных вариантов.

Известно, что при проектировании схемы символы берутся из встроенной, рабочей библиотеки проекта или из системных библиотек (урок 7). В библиотеках содержатся откомпилированные модули. Некоторые из них могут не иметь графического описания (вероятно, чтобы экономить память). В таких случаях для создания символа придется воспользоваться услугами специального редактора символов (**Symbol Editor**) или позволить редактору блок-схем (**Block Diagram Editor**) сделать это автоматически по стандартному шаблону.

Реальный символ будет сгенерирован в момент переноса его виртуального образа из «ящика символов» на схему. Генерация символа происходит только один раз при первом вызове нового компонента на схему.

Следует помнить, что если вы измените содержимое символа, то его внешний вид не будет обновлен автоматически. Это придется сделать вручную или использовать команду **Compare Interfaces**. Справедливо и обратное утверждение: изменения, внесенные в графику символа, не будут автоматически перенесены на его содержимое. Поэтому будьте внимательны и следите за соответствием символа его внутреннему содержанию.

Если вас устраивает стандартное изображение символа, автоматически генерируемое системой, то можно не беспокоиться о внешнем представлении компонента, достаточно создать только внутреннее описание символа, то есть его поведенческую или структурную модель.

Иногда имеет смысл выбрать компромиссный вариант: сначала создать внутреннее описание символа, потом получить от системы автоматически сгенерированное графи-

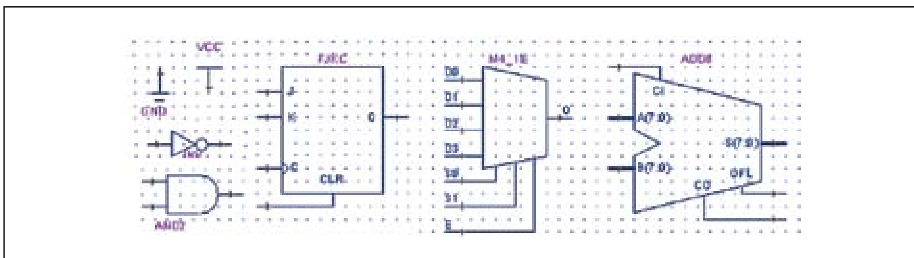


Рис. 1. Примеры символов из системной библиотеки Spartan3

ческое изображение и отредактировать его согласно своим вкусам и требованиям.

Технология создания символов

После небольшой теоретической разминки перейдем к практической работе и попробуем создать полноценный символ компонента с логической функцией **NAND2**. Назовем его **nand2_my**.

Проще всего выполнить внутреннее описание символа в текстовом **HDL**-формате (урок 5).

Создадим новый проект **Lesson_10**, активизируем закладку и запустим **HDE**-редактор. С помощью «мастера» **New Source File Wizard** введем основную информацию о будущем символе (рис. 2).

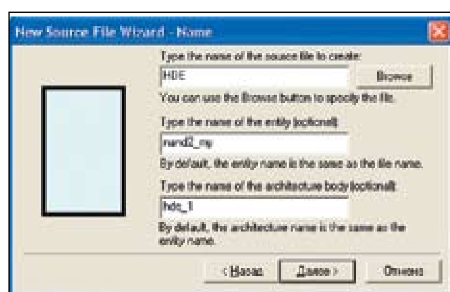


Рис. 2. Вводим основную информацию о будущем символе nand2_my

Итак, символ будет называться, как и объект проекта, **nand2_my**. Его внутреннее описание мы собираемся поместить в исходный файл **HDE.vhd** (отразив в названии инструмент создания — текстовый **HDE**-редактор).

Тем же именем назовем и архитектурное тело, добавив к нему номер версии будущей архитектуры **hde_1**. Позднее станет ясно, почему имя объекта проекта в нашем эксперименте не должно совпадать с названием архитектуры.

На следующей панели «мастера» зададим информацию о контактах символа (**I1**, **I2** — входы, **O** — выход). С этим процессом мы хорошо знакомы (урок 1 или 7), а потому оставим его без комментариев.

По окончании работы «мастера» получится **VHDL**-шаблон внутреннего описания символа, в который нужно добавить оператор (**O<=I1 nand I2 after 5 ns;**), реализующий функцию компонента **nand2_my** (рис. 3).

```
25  library IEEE;
26  use IEEE.STD_LOGIC_1164.all;
27
28  entity nand2_my is
29      port(
30          I1 : in STD_LOGIC;
31          I2 : in STD_LOGIC;
32          O : out STD_LOGIC
33          );
34  end nand2_my;
35
36  architecture hde_1 of nand2_my is
37  begin
38      O<=I1 nand I2 after 5 ns; -- добавленная строка
39  end hde_1;
```

Рис. 3. Внутреннее описание компонента nand2_my на языке VHDL

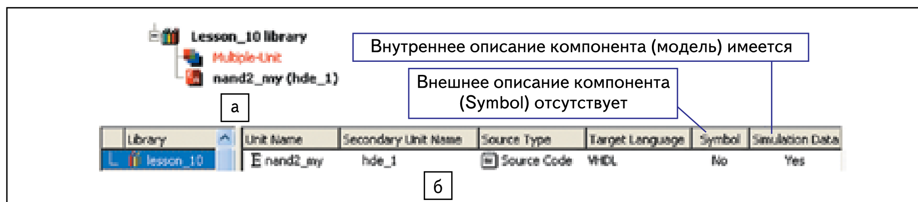


Рис. 4. В рабочей библиотеке lesson_10 появилось внутреннее описание компонента nand2_my. Графическое описание (Symbol) пока отсутствует

Откомпилируем созданный файл и посмотрим на содержимое рабочей библиотеки. Оно представлено в двух местах: в окне просмотра проекта (**Design Browser**) (рис. 4а) и на закладке менеджера библиотек (**Library Manager**) (рис. 4б).

Значок (**E/A** от слов **Entity/Architecture**) говорит о том, что в рабочей библиотеке появились два связанных модуля (**Unit**) — первичный модуль с именем **nand2_my** и вторичный (**Secondary Unit**) с именем **hde_1** со ссылкой на первый (строка 36 на рис. 3). Рис. 4б информирует нас о том, что в библиотеке имеются данные для моделирования (**Simulation Data = Yes**), но графического образа пока нет (**Symbol = No**).

Вы хотите посмотреть, как будет выглядеть символ? Тогда щелкните **ПКМ** на строке **nand2_my** (**hde_1**) (рис. 4а) и выполните команду **Edit Symbol**. Откроется окно редактора символов **Symbol Editor**, и в него загрузится предлагаемое системой по умолчанию графическое изображение компонента **nand2_my** (рис. 5а).

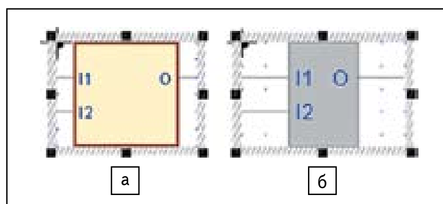


Рис. 5. Графическое изображение компонента nand2_my:
а) с параметрами, заданными по умолчанию;
б) с параметрами, измененными пользователем

Вы разочарованы? Не нравится желтый фон, толстая коричневая обводка? Большие габариты символа? Однако не спешите отказываться от системных услуг. Вы видите проявление настроек по умолчанию. Все это можно изменить.

На рис. 5б показан тот же символ с новыми настройками. Он тоже сгенерирован системой, но, согласитесь, мало похож на первый.

Чтобы задать желаемые параметры, активизируйте команду **Preferences** из меню **Tools**, в окне **Category** выберите категорию **Environment/Appearance** («Среда/вид»), а в поле **Windows** — схемный редактор (**Block Diagram Editor**). Вам понадобятся две подкатегории — **Symbol** и **Symbol name**. Установки последней категории проявляют себя только при размещении символа на схеме.

Для изменения размеров символа потребуется другая категория — **Block Diagram Editor/Symbols Generation**. Кроме габаритных размеров там можно скорректировать также длину выводов и расстояние (шаг) между ними.

Можно легко удалить символ и создать его вновь с измененными настройками. Для этого достаточно выделить его на закладке менеджера библиотек (рис. 4б), распахнуть контекстное меню и выполнить команду **Delete Symbol** (или нажать на кнопку инструментальной панели **Library Manager**).

Впрочем и удалять-то ничего не нужно, так как новые настройки начинают действовать немедленно. Более того, они распространяются не только на редактируемый объект, но и на все символы, ранее сгенерированные системой, в том числе и на символы из встроенной библиотеки **Built-in Symbols**.

Посмотрите на встроенный символ, показанный на рис. 6. Он выглядел совсем иначе, когда мы проектировали схему мультиплексора на первом уроке.

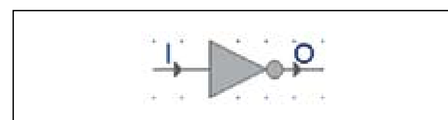


Рис. 6. Графическое изображение мультиплексора

Итак, если вас устраивает графическое изображение компонента на схеме, сгенерированное системой, то можно вообще забыть о существовании такой программы, как редактор символов (**Symbol Editor**). В этом случае задача построения библиотечных компонентов заметно упрощается и фактически сводится к созданию только внутреннего представления символа. Надеемся, что рассмотренный пример достаточно наглядно подтверждает правдивость этих слов.

Можем ли мы с такой же легкостью манипулировать данными для моделирования? В принципе при желании от них тоже нетрудно избавиться, но только с определенными издержками. Один способ состоит в том, чтобы отсоединить от проекта исходный файл с внутренним описанием компонента.

С этой целью в окне просмотра проекта выделим файл **HDE.vhd**, откроем контекстное меню и выполним команду **Remove**. Появится диалоговая панель с тем же названием (рис. 7).

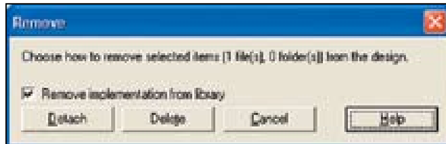


Рис. 7. Диалоговая панель Remove для удаления ресурса с диска (кнопка Delete) или отсоединения его от проекта (кнопка Detach)

Чтобы отсоединить исходный файл от проекта, не удаляя его с диска, следует нажать на кнопку **Detach** («Отсоединить»). Но сначала нужно убедиться, что установлен флажок **Remove implementation from library** («Удалить реализацию из библиотеки»). В противном случае в библиотеке сохранятся откомпилированные ранее данные для моделирования. При установленном флажке мы получим желаемый результат (рис. 8).

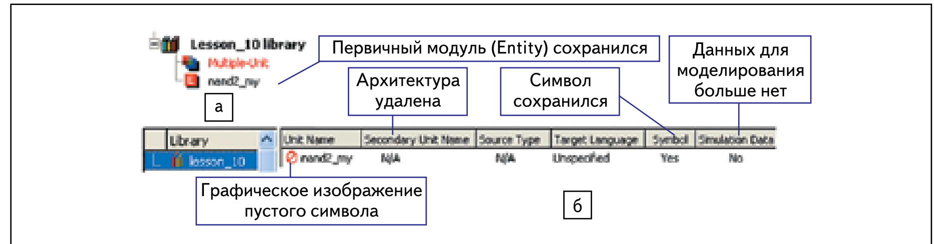
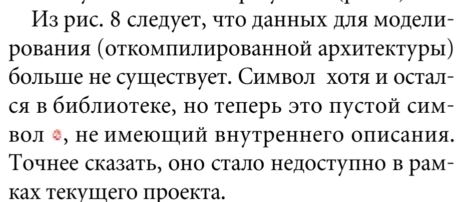


Рис. 8. Компонент nand2_my теперь не имеет данных для моделирования, символ лишился внутреннего описания и приобрел статус пустого символа

Из рис. 8 следует, что данных для моделирования (откомпилированной архитектуры) больше не существует. Символ хотя и остался в библиотеке, но теперь это пустой символ, не имеющий внутреннего описания. Точнее сказать, оно стало недоступно в рамках текущего проекта.



Второй способ избавиться от данных моделирования более грубый. В этом мы сейчас убедимся. Но сначала подключим файл **HDE.vhd** к проекту (команда **Add Files to Design** из меню **Design**) и вновь откомпилируем его.

Затем щелкнем **ПКМ** на строке **Lesson_10 library** в окне просмотра проекта (рис. 4а) и выполним команду **Delete simulation data**.

Увы, очистится все содержимое рабочей библиотеки, и вы потеряете все откомпилированные данные. Правда, потерянное легко восстановить, выполнив повторную компиляцию проекта (команда **Design/Compile All**).

Но чтобы исключить из процесса компиляции файл **HDE.vhd**, щелкните на нем **ПКМ** и выполните команду **Exclude from Compilation**. Вы увидите, что имя файла теперь записано курсивом. Так помечаются все файлы проекта, исключенные из компиляции.

Проектирование символов с несколькими реализациями

А теперь познакомимся с одной полезной особенностью **VHDL**-проектов, о которой упоминалось ранее. Речь пойдет о возможности задать для одного компонента несколько реализаций (**Symbols with Multiple Implementations**), а точнее — для одного объекта проекта (**entity**) построить несколько архитектур (**architecture**).

Практическая польза данного механизма заключается в том, что он позволяет постепенно, компонент за компонентом переходить от поведенческого описания проекта к его структурной реализации в процессе ручного высокоуровневого синтеза.

Первое, что приходит в голову, это открыть файл **HDE.vhd** (рис. 3) и дописать в него еще одну архитектуру, например **hde_2** (рис. 9). Проще всего скопировать уже имеющуюся архитектуру и внести в копию необходимые изменения. Копию вставим в конец файла. Как выяснится позднее, это важная оговорка.

Чтобы при моделировании компонента **nand2_my** понять, какая из двух архитектур окажется активной, нужно сделать так, чтобы они чем-то отличались. Функции у обеих архитектур одинаковые, значит, вся надежда на задержки. Поэтому зададим для второй архитектуры другую задержку, например **10 ns** (напомню, что у первой она равнялась **5 ns**).

Откомпилировав схемный файл, мы увидим в окне менеджера библиотек результат своей деятельности: с первичным модулем **nand2_my** теперь связаны *две* архитектуры — **hde_1** и **hde_2** (рис. 10).

А теперь попробуем ответить на самый интересный вопрос. Как связать объект проекта с одной из созданных архитектур? Другими словами, как задать конкретному компоненту активную архитектуру? Чтобы прояснить этот вопрос, создадим новый схемный файл **test_nand2_my** и разместим на схеме два экземпляра компонента **nand2_my** (рис. 11).

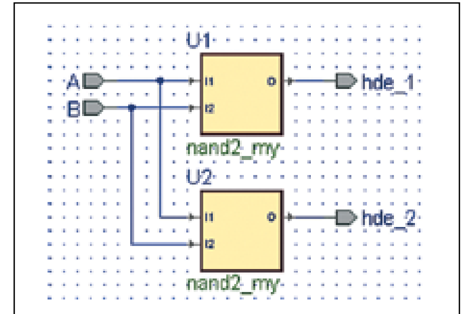


Рис. 11. Схема для тестирования созданного символа nand2_my с несколькими архитектурами

Щелкните дважды **ЛКМ** на каждом символе, и вы увидите в обоих случаях одну и ту же реализацию внутреннего описания — это архитектура **hde_2**. Именно ее отмечает мигающий курсор **HDE**-редактора.

Результат подтверждает уже известный тезис о том, что активной по умолчанию является архитектура, откомпилированная последней. А последним компилируется модуль, находящийся в конце файла **HDE.hdl**.

Откройте файл **HDE.hdl** и поменяйте местами архитектуры **hde_1** и **hde_2**. Откомпилируйте его заново и вновь посмотрите содержимое символов **U1** и **U2**. Теперь активной должна стать архитектура **hde_1**.

Убедившись, что установки по умолчанию работают, попробуем задать компонентам **U1** и **U2** разные активные архитектуры. Но сначала восстановим их прежнюю последовательность в файле **HDE.vhd** (сначала **hde_1**, потом **hde_2**).

Затем щелкнем **ПКМ** на верхнем символе **U1** и выполним команду **Properties**. Откроется панель **Symbol Properties** (рис. 12). В поле **Architecture**: выберем для него активную архитектуру **hde_1** и установим флажок **Generate configuration specification**.

Очень важный момент! *Данный флажок разрешает включить в генерируемый из схемы VHDL-файл объявление конфигурации, которая собственно и несет информацию об активной архитектуре.*

Для компонента **U2** активной сделаем архитектуру **hde_2**. Последнюю операцию выполнять необязательно, так как она повторяет установки, сделанные по умолчанию,

```
41   architecture hde_2 of nand2_my is
42   begin
43       O<=not (I1 and I2) after 10 ns; -- обновленная строка
44   end hde_2;
```

Рис. 9. В исходный файл HDE.vhd добавлена вторая архитектура — hde_2

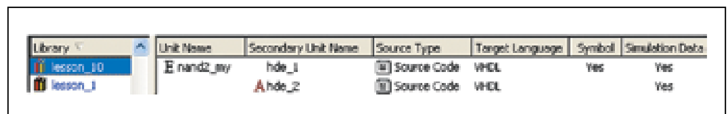


Рис. 10. В окно Library Manager добавилась вторая архитектура — hde_2

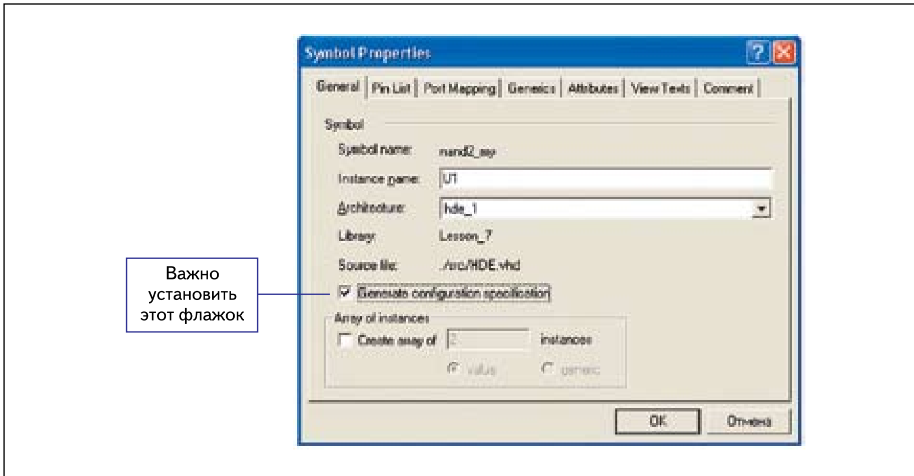


Рис. 12. С помощью диалоговой панели Symbol Properties задаем имя активной архитектуры и разрешаем сгенерировать данные о текущей конфигурации

```
46   ---- Configuration specifications for declared components
47   for U1 : nand2_my use entity work.nand2_my(hde_1);
48   for U2 : nand2_my use entity work.nand2_my(hde_2);
```

Рис. 13. Пример VHDL-кода

но нам заданная явно конфигурация понадобится при сравнении вариантов.

Для рассматриваемого примера сгенерированный из схемы **VHDL**-код будет включать спецификацию конфигурации в форме, представленной на рис. 13.

Здесь сообщается, что для (**for**) компонента **U1** должен использоваться (**use**) объект проекта (**entity**) с именем **nand2_my**, находящийся в рабочей библиотеке (**work**), с активной реализацией **hde_1**. Для компонента **U2** — все то же самое, но активной является другая архитектура с именем **hde_2**.

Как видно, активная реализация задается в конце каждой строки, а ее имя объявляется в круглых скобках. Понизьте уровень описания компонентов **U1** и **U2**, и вы увидите, что механизм конфигурации исправно выполняет свою функцию.

Мигающий курсор **HDE**-редактора в первом случае устанавливается на архитектуре **hde_1**, а во втором случае — на архитектуре **hde_2**.

Чтобы избавиться от последних сомнений, промоделируем схему, показанную на рис. 11. Напомним, что в архитектуре **hde_1** была установлена задержка в 5 ns, а для архитектуры **hde_2** — 10 ns (рис. 3 и 9). Названные задержки хорошо видны на графиках для выходных сигналов **hde_1** и **hde_2** (рис. 14).

Вам кажется, что основные трудности уже позади? Тогда попробуйте присоединить к символу **nand2_my** не текстовое, а графическое содержимое, например схему или диаграмму состояний.

Оказывается, нужно щелкнуть **ПКМ** на строке **nand2_my** (**hde_1**) или **nand2_my** (**hde_2**) в окне **Design Browser** и выбрать команду **Add**

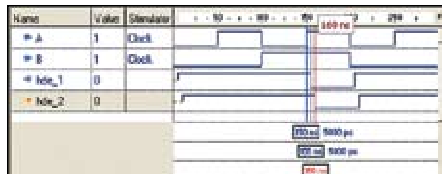


Рис. 14. Результаты моделирования схемы для тестирования символа nand2_my

New Architecture. Откроется диалоговая панель **Create New Architecture** (рис. 15), и вы увидите все возможные варианты реализации.

Правда, выбор невелик, вариантов всего три: архитектуру можно получить из схемы, диаграммы состояний или непосредственно в текстовом **HDE**-редакторе.

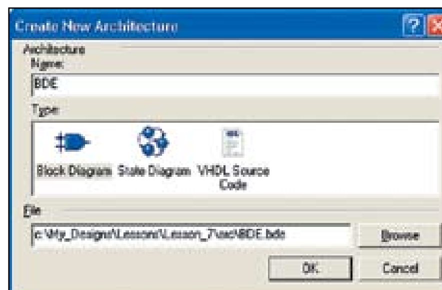


Рис. 15. Active-HDL позволяет задать новую архитектуру тремя способами, сгенерировав ее из схемы, диаграммы состояний или непосредственно в текстовом HDE-редакторе

Диаграмму состояний мы пока не умеем проектировать, поэтому остановимся на схеме. Введем имя архитектуры, например, **BDE** (от слов **Block Diagram Editor**), выберем нужный тип описания (он задан по умолчанию) и нажмем кнопку **OK**. В окне просмотра проекта появится новый файл .

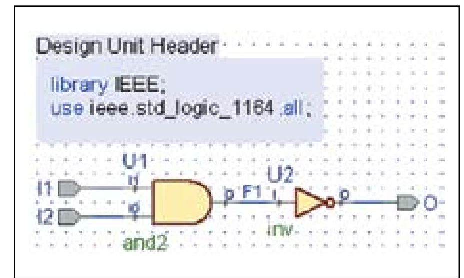


Рис. 16. Схемное представление компонента nand2_my

Откроем его и нарисуем простую схему, реализующую операцию **NAND2** (рис. 16). Изначально на схеме появятся порты **I1**, **I2**, **O** и текстовый **HDL**-блок **Design Unit Header** («Заголовок модуля проекта»). Их система размещает автоматически.

С текстовыми **HDL**-блоками мы уже познакомились на уроке 9. Поэтому остается только добавить символы **and2** и **inv**.

Прежде чем пойти дальше, откройте диалоговую панель **Code Generation Settings** (команда меню **Diagram/Code Generation Settings**) и убедитесь, что система собирается генерировать только архитектуру (установлен флажок **Generate Architecture Only**). Посмотрите, для какого объекта проекта она будет генерироваться. В поле **Entity** должно стоять его имя — **nand2_my**. Если все правильно, самое время нажать на кнопку **OK**.

Теперь надо преобразовать схемное описание в **VHDL**-код. Система выполняет эту операцию в автоматическом режиме, отсюда и ее название — **Generation** («Генерация»). Подходящая команда обнаруживается в нескольких местах. Мы воспользуемся самым простым способом: щелкнем по пиктограмме **Generate HDL Code** . Результат будет записан в файл . Его содержимое подтверждает, что сгенерирована только архитектура с именем **BDE** (рис. 17).

```
22   library IEEE;
23   use ieee.std_logic_1164.all;
24
25   architecture BDE of nand2_my is
26       signal F1 : STD_LOGIC;
27   begin
28       F1 <= I2 and I1;
29       O <= not(F1);
30   end architecture BDE;
```

Рис. 17. Результат автоматической генерации архитектуры BDE для компонента nand2_my

Осталось сделать последний шаг: откомпилировать полученный **VHDL**-код. После компиляции в рабочей библиотеке проекта должна появиться новая строчка , а в окне менеджера библиотек — еще одна архитектура **bde** (рис. 18).

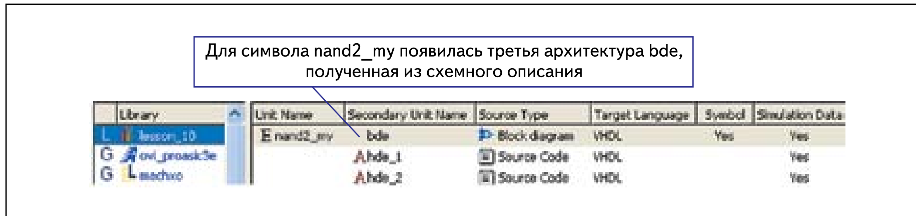


Рис. 18. Символ nand2_my получил еще одну архитектуру — bde, полученную из схемного описания (Block diagram)

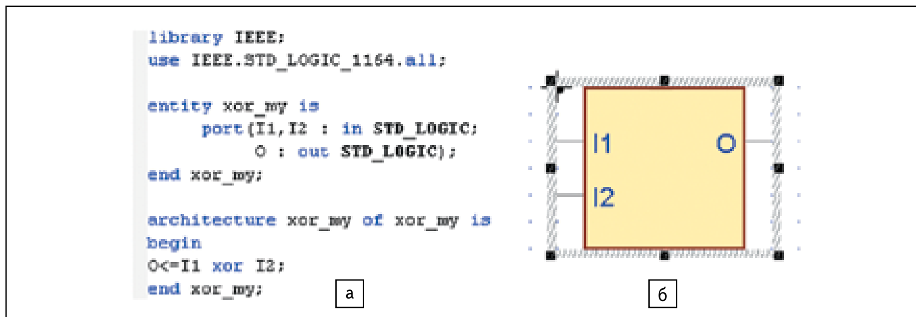


Рис. 19. Внутреннее а) и внешнее б) описания компонента xor_my

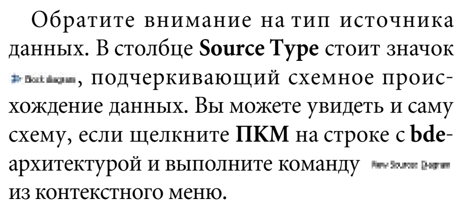
Создадим внутреннее описание компонента **xor_my** и поручим системе сгенерировать его графический образ (рис. 19). Не выходя из графического редактора **Symbol Editor**, вызовем диалоговую панель **Symbol Properties** и активизируем закладку **Generics** (рис. 20). Введем имя настраиваемого параметра **TP** (от слов **Time Propagation** — «Задержка распространения»), укажем его тип (**TIME**) и значение по умолчанию, например **1ns**.

Понятно, что подобным образом можно сформировать целый список настраиваемых параметров, но нам достаточно одного. Символ нужно обязательно сохранить.

А теперь самое интересное. Для созданного символа активизируем команду **Compare Symbol with Contents** («Сравнить символ с его содержимым») из меню **Symbol**.

На первой закладке **Ports** несоответствий нет: количество и имена контактов символа совпадают с аналогичными параметрами портов в его модели (рис. 21).

Однако в левом нижнем углу панели мы обнаруживаем сообщение об одном несовпадении на закладке **Generic**. Откроем ее, чтобы убедиться, в чем конкретно оно состоит (рис. 22): в содержимом символа отсутствуют данные о настраиваемом параметре **TP**.



Обратите внимание на тип источника данных. В столбце **Source Type** стоит значок Block diagram, подчеркивающий схемное происхождение данных. Вы можете увидеть и саму схему, если щелкните **ПКМ** на строке с **bde**-архитектурой и выполните команду Rev Source Diagram из контекстного меню.

Предлагаем самостоятельно проверить, что у нас получился полноценный символ, содержимым которого является схема.

Добавьте в файл **test_nand2_my.bde** еще один символ **nand2_my** и раскройте его содержимое командой **Push**. Должна открыться схема, ведь она компилировалась последней.

Промоделируйте проект, чтобы убедиться, что сигнал на выходе третьего компонента появляется без задержки. Отсутствие задержки является особенностью элементов из встроенной библиотеки **Built-in Symbols**, которые использовались при проектировании нашей схемы **bde.bde** (рис. 16).

Настраиваемые параметры компонентов

Язык **VHDL** дает пользователю еще одну великолепную возможность — индивидуально настраивать для каждого символа, размещенного на схеме, его параметры. И что самое ценное, выполняется такая настройка извне, за пределами внутреннего описания символа.

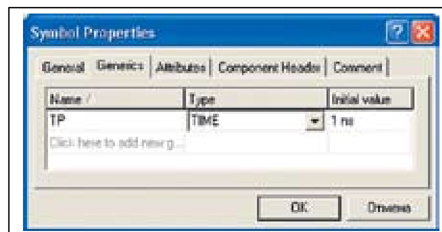


Рис. 20. Задаем имя, тип и значение по умолчанию настраиваемого параметра TP

Для задания настраиваемых параметров объекта проекта (**entity**) употребляется ключевое слово **generic**. Хотя настройки и не видны на графическом изображении символа, но их можно посмотреть и изменить с помощью диалоговой панели **Symbol Properties**.

Очень часто в качестве настраиваемого параметра выступает задержка компонента. И сейчас мы посмотрим, как можно имитировать естественный (технологический) разброс данного параметра.

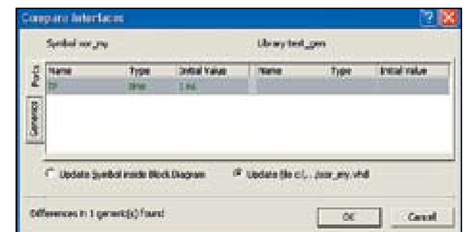


Рис. 22. В исходном файле xor_my.vhd отсутствует информация о настраиваемом параметре TP

Система предлагает привести символ и его содержимое в полное соответствие, путем изменения исходного файла **xor_my.vhd** (переключатель установлен в положение Update file c:\...\xor_my.vhd). Это разумное предложение, и с ним нельзя не согласиться. После обновления в файл автоматически добавится объявление настраиваемого параметра **TP** (рис. 23, строка 28).

Чтобы увидеть, как работают настраиваемые параметры, проведем модельный

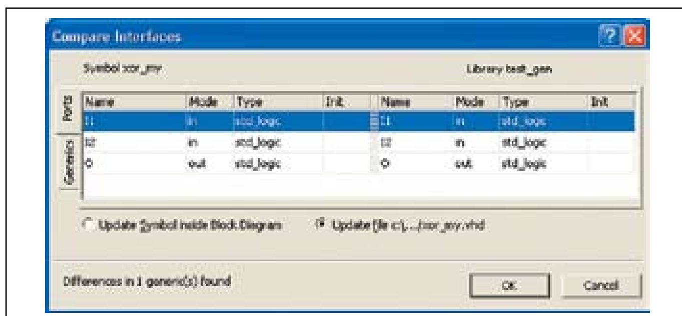


Рис. 21. В описании контактов символа и портов в его модели расхождений нет

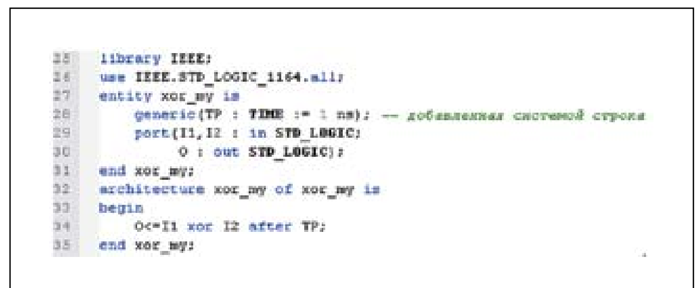


Рис. 23. В исходный файл xor_my.vhd автоматически добавилось объявление настраиваемого параметра TP

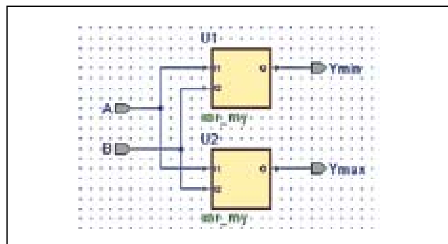


Рис. 24. Схема для тестирования индивидуальных задержек на компоненте xor_my

эксперимент. Создадим схемный файл **test_generic.bde** с двумя экземплярами символа **xor_my** (рис. 24). Одному зададим минимальную задержку, например, **5 ns**, другому — максимальную, например **15 ns**.

Выполняется такая работа очень просто. Для каждого символа отрывается диалоговая панель **Symbol Properties** и активизируется закладка **Generics**. В поле **Actual value** задается желаемое (действительное) значение задержки: **5** и **15 ns** соответственно (рис. 25).

Результаты моделирования схемы **test_generic.bde** (рис. 26) показывают, что теперь каждый экземпляр компонента **xor_my** имеет свою индивидуальную задержку, приобретенную благодаря настраиваемому параметру **TP**.

Рассмотренный столь подробно механизм настраиваемых параметров **generic** трудно переоценить. Он упрощает освоение современных технологий проектирования электронной аппаратуры, которые предполагают многократное (повторное — **reuse**) использование ранее разработанных моделей в новых проектах. А для того чтобы их можно было приспособить к условиям конкретного применения, они должны быть адаптируемыми или настраиваемыми.

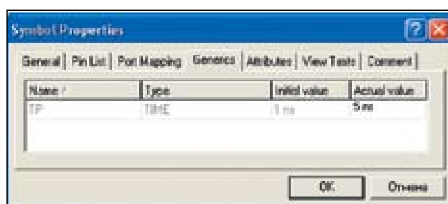


Рис. 25. В столбце Actual value задаем действительную задержку каждому экземпляру компонента xor_my

Настраиваемые параметры — это не только задержки, рассмотренные в примере. Такие параметры используются также для параметризации разрядности шин, числа портов и т.п.

Реализация настроек выполняется конструкцией **generic map**, входящей в оператор конкретизации компонента.

Посмотрим на фрагмент **VHDL**-кода, сгенерированного системой по схемному описанию из файла **test_generic.bde**. Для экономии места он подвергся небольшой правке (рис. 27). Здесь хорошо видно, что экземпляры компонента **xor_my** с метками реализации **U1** и **U2** получили индивидуальные задержки: **5** и **15 ns** соответственно.

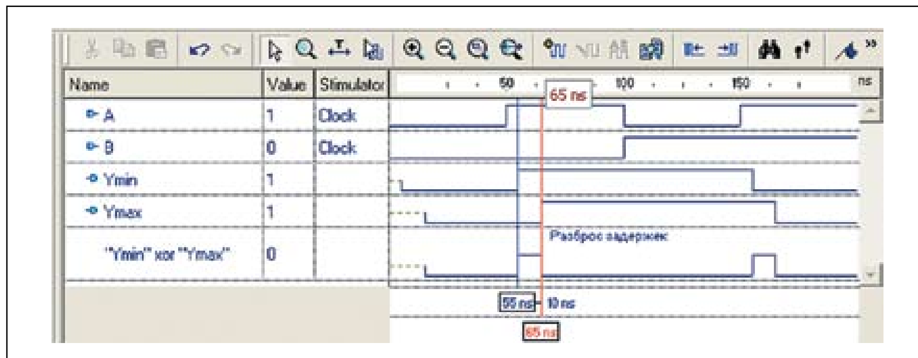


Рис. 26. Два экземпляра одного и того же компонента xor_my имеют разные задержки

```
34  architecture test_g of test_g is
35      component xor_my
36          generic(
37              TP : TIME := 1 ns);
38          port (I1,I2 : in STD_LOGIC;
39               O : out STD_LOGIC);
40      end component;
41
42  begin
43      U1 : xor_my
44      generic map (TP => 5 ns)  -- Задержка 5 ns
45      port map(I1 => A,I2 => B,O => Ymin);
46
47      U2 : xor_my
48      generic map (TP => 15 ns) -- Задержка 15 ns
49      port map(I1 => A,I2 => B,O => Ymax);
50  end test_g;
```

Рис. 27. Реализация значений настраиваемых параметров осуществляется конструкцией generic map (строки 44 и 48)

Проектирование собственных символов

Заканчивая урок, отдадим дань людям с художественными наклонностями, то есть тем, кто желает создать свой собственный символ, как говорится, «с нуля».

Чтобы начать работу с графики символа, нужно отыскать в левой части меню **Standard** стрелку с выпадающим списком новых ресурсов и выбрать кнопку **New Symbol** (рис. 28).

Рис. 28. Выбор кнопки New Symbol

Вы сразу попадете в окно графического редактора **Symbol Editor** с предлагаемой по умолчанию заготовкой будущего символа (рис. 29а).

Заготовка представляет собой коричневый прямоугольник с заливкой желтого цвета, вписанный в контур символа с серой штриховкой. Контур определяет границы, в пределах которых должна размещаться графика символа.

Черные узелки на контуре говорят о том, что его размеры можно изменять. Потяните за любой из них, и вы увидите, что символ повторяет размеры контура (рис. 29б).

Кстати, на рис. 29б проявилась точка привязки, в которую помещается курсор мыши при размещении символа на схеме. По умолчанию она находится в левом верхнем углу габаритного прямоугольника, и поэтому была практически незаметна (рис. 29а).

А теперь проделаем маленький, но очень показательный эксперимент. Нарисуем внутри габаритного прямоугольника какой-нибудь графический объект, например, квадрат (рис. 29в), и вновь изменим размеры символа. Заметили? Новый объект почему-то не чувствителен к операции масштабирования. Он сохраняет не только свои размеры, но и местоположение.

Попробуйте удалить квадрат. Никаких проблем не возникает. А прямоугольник удаляется? Нет. Значит, у них разные свойства. И все зависит от того, является графический объект элементом фоновой графики или нет.

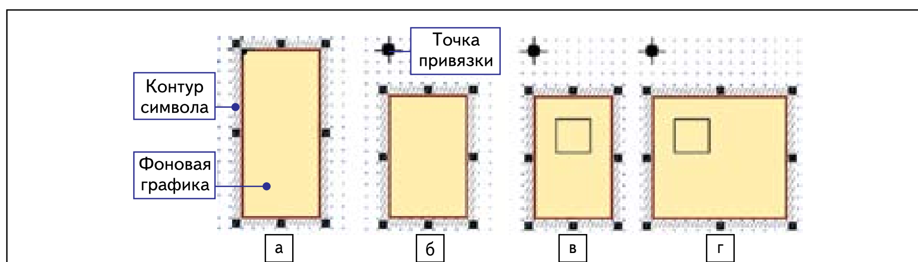


Рис. 29. Объекты фоновой графики (Symbol Background) повторяют размеры контура символа. Элементы не фоновой графики (квадрат) не чувствительны к масштабированию

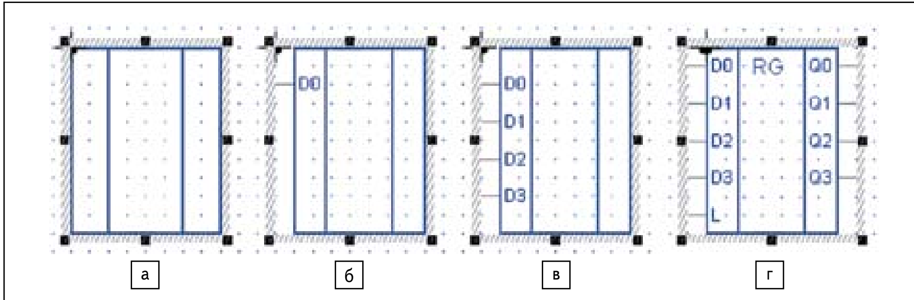


Рис. 30. Этапы создания символа

Рис. 31. Инструментальный ящик с контактами символа

Удалим все объекты так, чтобы контур символа стал пустым, и начнем проектирование нового символа с «чистого листа». Сначала нарисуем графику символа (рис. 30а) и сделаем ее фоновой.

Затем щелкнем по пиктограмме **Pins Toolbox** и вызовем инструментальный ящик с контактами (рис. 31).

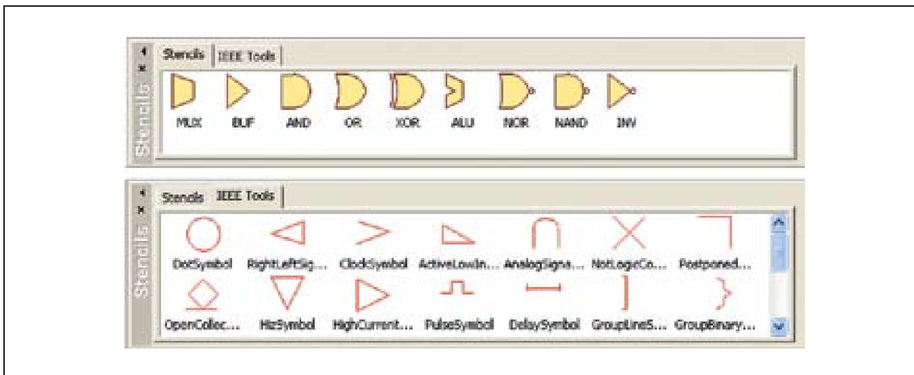


Рис. 32. Системные закладки ящика трафаретов Stencils

Вызовем для квадрата контекстное меню и выполним команду **Symbol Background**. Мы сделали квадрат элементом фона, и теперь он будет масштабироваться вместе с телом символа.

Чтобы появилась возможность редактировать фоновые объекты, нужно щелкнуть **ПКМ** в любом месте рабочей области редактора символов и активизировать команду **Edit Symbol Background**.

Контакты добавляются к проектируемому символу методом **Drag and Drop**. Для этого надо «прижать» нужный контакт мышью и отбуксировать его к символу. Перемещаемые контакты автоматически привязываются к контуру символа (с внутренней стороны), уменьшая размеры самого символа.

При вводе группы однородных контактов, например, **D0…D3** или **Q1…Q7**, удобно пользоваться командой **Duplicate** из контекстного меню. Отредактировав первый введенный контакт (рис. 30б), вызываем диалоговую панель **Duplicate** и задаем желаемое количество их «клонов». Если контактов много, можно уменьшить расстояние между ними. Все остальное будет сделано автоматически (рис. 30в).

Для повторного использования графики созданного символа (рис. 30г) вы можете сохранить ее как шаблон в инструментальном ящике трафаретов **Stencils** («Шаблон, образец, трафарет»).

Ящик трафаретов вызывается щелчком по пиктограмме с тем же названием. Изначально он имеет две закладки (рис. 32): одна называется **Stencils** и содержит нестандартные графические шаблоны, которые трудно нарисовать, другая носит имя **IEEE Tools** и включает условные обозначения, добавляемые к символу в стандарте **IEEE**.

Системные закладки доступны только для чтения. Из них можно перенести шаблон на проектируемый символ, но добавить сюда ничего нельзя. Поэтому придется создавать свою (пользовательскую) закладку. Щелкнем **ПКМ** в окне **Stencils** и выполним команду **New Page**.

Появится новая закладка **Stencil1**, которую мы тут же переименуем, например в **Stencil_my**. Здесь можно сохранять собственные разработки. Но предварительно полезно сгруппировать все графические объекты, передаваемые в библиотеку трафаретов. Выделим на символе все фоновые объекты и объединим их в одно целое командой контекстного меню **Group**. Осталось только отбуксировать фоновую графику символа на закладку **Stencil_my**.

Созданному шаблону можно присвоить имя, например, **RTL**, и использовать его при проектировании новых символов с похожей графикой, например счетчиков. Проделайте эту работу самостоятельно и убедитесь в том, что шаблоны экономят ваше время, особенно, если они имеют нестандартную форму, как в зарубежных обозначениях.

Перенеся шаблон на рабочую область редактора символов, не забудьте присвоить ему статус элемента фона, то есть установить для него флажок **Symbol Background**.

По умолчанию символы сохраняются в рабочей библиотеке проекта. Но у вас есть возможность записать символ и в отдельный файл, который получит расширение ***.bds**. Для этого достаточно выполнить команду **Save to File As…** из меню **File**.

Наконец, вы можете сохранить символ в любой другой доступной для записи библиотеке, например в своей собственной библиотеке **my_lib**. Инициируйте команду **Save to Library As…** из меню **File**, выберите из списка нужную библиотеку и нажмите кнопку **Save**. Иногда бывает полезно сохранить символ под другим именем. ■

Продолжение следует