Продолжение. Начало в № 3 ’2008

Краткий курс HDL. Часть 3. О написании кода «вообще»...

Иосиф КАРШЕНБОЙМ
iosifk@narod.ru

В этом разделе будут обсуждены вопросы, связанные с «художественным оформлением текстов». На первый взгляд, проблема мелкая. Но это только на первый взгляд. Да и то, только на очень неопытный взгляд...

Оформление текстов

Представим, что вы начали новый проект. Уже получено техническое задание и определены контуры будущего изделия. Можно приступать к работе? Да, конечно. Но только сначала давайте ответим на один, но довольно существенный, вопрос: «Сколько лет будет жить разрабатываемый вами проект?». Ответов может быть два. Первый ответ таков. Проект выполнен, запущен в производство. Следовательно, на этом жизнь проекта закончена и никаких доработок и исправлений в него вноситься никогда не будет. Не будет и модернизации этого проекта в следующих изделиях. В реальной жизни такие проекты — редкость. Обычно это малобюджетные разовые поделки. И к тому же очень простые поделки. Поэтому такие проекты обычно выполняются только одним разработчиком, они нигде не хранятся и не сопровождаются, и к таким проектам нет никаких требований по оформлению. В дальнейшем такие проекты мы обсуждать не будем. Но также известен и другой ответ на поставленный выше вопрос — проект будет жить ровно столько, сколько лет будет эксплуатироваться изделие, в состав которого он входит. К таким проектам относятся все проекты для промышленности, поскольку там изделия эксплуатируются десятки лет. И в течение всего срока эксплуатации изделия может потребоваться ревизия проекта. Следовательно, проект должен сопровождаться. Что это значит? Вы только что написали файл. Прекрасно! Вы точно знаете, что сигнал «a», например, — это выходная шина, а сигнал «z» — это шина управления. А вспомните ли вы названия и выполняемые функции для каждого из сигналов через три месяца, через полгода, через три-четыре года? Скорее всего, нет, потому что при таких обозначениях сигналов в них нет информации о выполняемой данным сигналом функции. Да и сам файл, который вы нашли в архиве проекта, — относится ли он к данному проекту вообще или этот файл попал туда случайно? А теперь представьте, что сопровождением занимается другой человек, тот, который не разрабатывал данный проект, и он вообще не знает о том, какие файлы входят в проект и как работает та или иная часть проекта. Отсюда выводы: проекты необходимо выполнять таким образом, чтобы они были документированы и легко сопровождались.

По вопросам оформления документации существует достаточно много рекомендаций, выработанных фирмами — изготовителями программного обеспечения. Здесь же будут изложены только самые общие принципы.

Первое, что необходимо сделать, это, конечно, разработать техническое задание. Из документов технического задания должны быть порождены следующие документы:

- разбивка проекта на модули, узлы, файлы;
- файлы проекта;
- техническое описание изделия;
- документы по проверке и тестированию изделия.

Причем хочется заметить, что в идеальном случае все эти документы должны выполняться одновременно. Исключение может быть сделано только для этапа разбивки проекта на части. Этот этап может быть выполнен с опережением по времени относительно других документов. Что отсюда следует? Вывод очевиден и прост. В самом начале проекта должна быть организована единая система архивирования версий. Должен быть назначен сотрудник, ответственный за полноту комплекта документации. Кроме того, что версии документов проекта должны попадать в соответствующие разделы архивов, сами по себе архивы должны быть выполнены так, чтобы не произошло потери информации. Резервные диски, архивация на DVD и так далее. Способов здесь много, и выбор конкретного способа зависит от объема проекта и от числа задействованных в нем участников.

К каждой версии документов необходимо иметь описание того, какие документы входят в состав комплекта на текущем этапе, что делает эта версия, чего не хватает до окончания этапа работ.

Теперь переходим к рассмотрению оформления отдельного файла проекта.

Общие рекомендации по оформлению файлов

Об использовании пробелов и табуляции

Правила выполнения текстов могут быть взяты из любого учебника по программированию [1]. В этой книге довольно подробно описано то, как правильно оформленный текст помогает избежать ошибок в проекте. Здесь приведены несколько рекомендаций по оформлению текстов и примеры для языка Си, но все сказанное здесь будет справедливо и для языка Verilog.

Весь текст должен быть выполнен с учетом отступов и табуляции. Комментарии должны быть выровнены вертикально (пример 1).

```
/* Первая строка,
 * вторая строка,
 * третья строка.
 */
// Первая строка,
// вторая строка,
// третья строка.
```

Пример 1. Комментарии, выровненные по вертикали

При написании текстов необходимо использовать выравнивание текста по столбцам везде, где это возможно.

Два следующих блока, приведенных в примере 2 функционально одинаковы, но заметьте, насколько легче искать имена переменных во втором блоке, причем не только из-за выравнивания комментариев, но и потому, что имена образовали аккуратный столбец.

Вариант текста без форматирования:

```
int x; // Опишите, что делает x.
unsigned long int (*pfi)(); // Опишите, что делает pfi.
const char *the_variable; // Опишите, что делает the_variable.
int z; // Опишите, что делает z.
x = 10; // Здесь идет комментарий.
the_variable = x; // Здесь второй комментарий.
z = x; // А здесь третий.
```

Вариант текста с форматированием:

```
Int             x;              // Опишите, что делает x.
unsigned long   int ( *pfi )(); // Опишите, что делает pfi.
Int             z;              // Опишите, что делает z.

const char      *the_variable;  // Опишите, что делает
                                // the_variable.
X               = 10;           // Здесь идет комментарий.
the_variable    = x;            // Здесь второй комментарий.
Z               = x;            // А здесь третий.
```

Пример 2. Варианты текста без форматирования и с форматированием

Не располагайте комментариев между именем функции и открывающей скобкой.

Ниже показано неправильное выполнение комментария (пример 3).

```
foo( int x )
/* Не помещайте
 * комментарий
 * здесь. */
{
        //...
}
```

Пример 3. Вариант текста с неправильно расположенными комментариями

Поместите этот комментарий или до имени функции, или вставьте его в тело функции ниже открывающей скобки (пример 4).

```
/* Или помещайте
** его здесь.
*/

foo( int x )
{
    /* или здесь,
    ** с таким же отступом, что и у кода.
    */
}
```

Пример 4. Вариант текста с комментариями

Еще один технологический прием заключается в том, что надо помечать конец длинного составного оператора чем-нибудь, имеющим смысл. Прежде всего, подобные комментарии в конце блока (пример 5) не дают

```
while( a < b )
{
for( i = 10; --1 >= 0; )
    {
        f( i );
    } // for
} // while
```

Пример 5. Вариант текста с комментариями

```
На первой странице:
   while( a < b )
   {
      while( something_else() )
      {
         for( i = 10; --1 >= 0; )
         {
            for( j = 10; --j >= 0; )
            {
   // далее идет масса кода
А на какой-то из последующих страниц:
            } // for
         } // for
      } // while
   } // while
```

Пример 6. Вариант текста с комментариями в конце блока

ничего, кроме неразберихи, если блоки короткие. Их используют только тогда, когда составной оператор слишком велик, чтобы поместиться на экране, или в нем столько уровней вложений, что невозможно понять суть происходящего. Комментарии в конце блока обычно целесообразны в больших составных операторах, но можно часто встретить подобный код (пример 6).

Эти комментарии слишком кратки, чтобы быть полезными. Завершающие блок комментарии должны полностью описывать управляющий оператор. Завершающие блок комментарии из предыдущего примера должны выглядеть таким образом, как показано в примере 7.

```
} //       for( j = 10; --j >= 0; )
          }  //  for( i = 10; --1 >= 0; )
      }          //  while( something_else() )
}                //  while( a < b )
```

Пример 7. Вариант текста с комментариями

Так же и для **#ifdef**. Так как он почти всегда находится на некотором расстоянии от **#endif**, то всегда целесообразно ставить метку у **#endif** (пример 8).

```
#ifndef __SOMEFILE_H_
#define __SOMEFILE_H_

// здесь следует 1000 строк программы

#endif // __SOMEFILE_H_
```

Пример 8. Вариант текста с меткой у #endif

То же самое относится и к **#else**.

Располагайте в строке только один оператор. Нет абсолютно никакой причины упаковывать в одну строку много операторов, только если у вас нет намерения сделать программу нечитаемой. Очевидным исключением является оператор for, все три части которого должны быть на одной строке.

Используйте штриховую линию для зрительного разделения фрагментов текста.

Это выглядит как строка с комментарием:

```
//-----------------
```

Штриховая линия между фрагментами текста облегчает чтение файла. Так же как пустая строка указывает на границу абзаца, штриховые линии подобны заголовкам разделов. Если же нужно еще более четкое разделение, то рекомендуется использовать:

```
//====================
//ОПИСЫВАЮЩИЙ ТЕКСТ
//====================
```

При этом вот как может выглядеть описание функции (пример 9).

Пробел — один из наиболее эффективных комментариев, и это может чрезвычайно улучшить читаемость программы. Пустые

```
//-------------------------------------------------------
void descriptive_name( type descriptive_name )
{
// Если имена функции и аргументов недостаточно содержа-
// тельны, то здесь должен быть помещен комментарий,
// описывающий то,что делает функция.
// Можно опустить комментарии, если имена
// достаточно понятны. (Соответствующее правило гласит:
// «Не объясняй очевидного»).
//
// Затем описываются возвращаемое значение и аргумент.
// И вновь можно не использовать комментарий, если
// имена достаточно удачные.
//
// Наконец, здесь помещается комментарий, описывающий, как
// функция делает то, что она делает. И снова опустить
// комментарии, если программа сама по себе достаточно
// содержательна.

    code_goes_here();
}
```

Пример 9. Вариант текста описания функции

строки (или отступ в первой строке) зрительно разделяют абзацы. Разбивайте текст программы на логические куски (то есть абзацы), где каждый кусок выполняет одну операцию. Окружите эти куски пробелами или пустыми строками.

За знаком препинания всегда должен идти пробел.

Операторы являются сокращениями слов. Когда вы видите «+», то говорите «плюс». Подобно любому сокращению, вы должны окружать идентификатор пробелами. (Например: a + b читается «a плюс b», a+b читается «аплюсb»).

Комментарии должны иметь тот же отступ, что и окружающий текст программы.

Абзацные отступы предназначены для того, чтобы сделать структуру вашей программы легко понятной. Если вы организуете отступы в комментариях беспорядочным образом, то этим вы лишите их смысла. Комментарий в следующей программе должен быть снабжен отступами (пример 10):

```
f()
{
/* Здесь идет
 ** длинный комментарий
 */
    code();
}
```

Пример 10. Комментарий в программе должен быть снабжен отступами

Выравнивайте скобки вертикально по левой границе. Иногда поиск отсутствующей фигурной скобки превращается в крупную проблему. Если вы вынесете скобки туда, где их хорошо видно, то их отсутствие будет сразу же заметно (пример 11):

```
while ( some_condition )
{
    // внутренний блок
}
А в стиле Кэрнигана и Ричи это не так очевидно:
if( condition ){
    code();
}else{
    more_code();
}
```

Пример 11. Выравнивайте скобки вертикально по левой границе

Если сравнить два фрагмента текста в примере 11, то становится видно, что в стиле Кэрнигана и Ричи не только трудно проверить скобки на парность, но и отсутствие зрительного разделения за счет строк, содержащих лишь открытые скобки, ведет к ухудшению читаемости.

Теперь снова вернемся к проектированию на HDL и рассмотрим правила оформления файлов, но уже с точки зрения разработки проектов

Как вы знаете, любой машиностроительный чертеж в правом нижнем углу имеет штамп, в котором помещены сведения о том, «что, кто, как, для чего и когда исправлял». И ни один инженер-конструктор никогда не будет вам говорить, что это только «украшение», а машину или агрегат и без этого собрать можно… Наоборот, за много лет практики система документооборота машиностроительных предприятий выработала у инженеров-конструкторов твердую уверенность в необходимости подобного рода документов. А вот начинающие разработчики такого мнения зачастую не имеют.

При проектировании на HDL в самом начале файла помещают служебную информацию, аналогичную той, что находится в штампе машиностроительного чертежа (пример 12). Там помещаются сведения о фирме, проекте, о том, кто вел разработку файла, кто проверял данный файл, о дате начала проекта и т. д. Объем информации и ее внешний вид не стандартизированы и в большинстве случаев определяются правилами оформления документации, принятыми в той или иной фирме.

```
//////////////////////////////////////////////////////////////////
// Name File      : STATMACH.v                                  //
// Autor          : Iosif Karshenboim                           //
// Company        :                                             //
// Description    : Timer, Width=4 bits   and FSM               //
// Start design   : 16.10.2003                                  //
// Last revision  : 16.10.2003                                  //
//////////////////////////////////////////////////////////////////
```

Пример 12. Заголовок файла проекта, минимальная информация о проекте и авторе

Да, действительно, часть проекта, выполненная в данном файле, будет работать и без заголовка файла, но как потом собрать такой проект целиком? Как сопровождать такой проект? В качестве еще одного примера хочется привести один из файлов процессора microblaze, выполненного на фирме Ксайлинкс (пример 13). В этот заголовок файла помещена следующая информация:

1) дата и время генерации версии файла;
2) название файла;
3) описание файла;
4) структура;
5) история изменений: кто, что и когда менял;
6) соглашение по именам сигналов.

Все пункты, кроме двух последних, уже обсуждались. Два же последних пункта нуждаются в дополнительных пояснениях. История изменений очень важна. Особенно в том случае, когда проект делается группой разработчиков. Представьте, что вы используете библиотечные файлы, в которые кто-то внес «кое-какие улучшения», не поставив в известность об этом своих коллег. Представьте также, что сегодня ваш проект почему-то стал работать «немного не так», и вы в шоке. Что случилось? Ведь вы ничего не меняли, и еще вчера все было хорошо… Таким образом, любые изменения, касающиеся «файлов общего пользования», должны быть задокументированы и разработчики, участвующие в общем проекте, должны быть поставлены в известность о том, что произошли изменения в файлах. Запись о том, что же конкретно менялось, поможет выявить ошибку, к которой это изменение может привести. И само внесение изменений в библиотечные файлы должно производиться только специально выделенным сотрудником. Так, например, каждый из разработчиков вносит изменения в свои файлы. Но до определенного момента действует старая версия файлов, и все участники проекта работают с этой версией файлов, проверенной на предыдущем этапе. Затем идет извещение о замене файлов, и старые файлы централизованно меняются на новые. После этого начинается проверка того, что при замене файлов не произошло ошибок и все остальные файлы проекта, в которых задействованы библиотечные файлы с изменениями, работают без ошибок. Только после этого разработчики приступают к дальнейшей работе над своими файлами.

```
--SINGLE_FILE_TAG
-------------------------------------------------------------------------------
-- $Id: shift_logic.vhd,v 1.3 2005/02/10 11:09:40 goran Exp $
-------------------------------------------------------------------------------
-- Shift_Logic — entity/architecture
-------------------------------------------------------------------------------
--                  ****************************
--                  ** Copyright Xilinx, Inc.  **
--                  ** All rights reserved.    **
--                  ****************************
-------------------------------------------------------------------------------
-- Filename:       shift_logic.vhd
-- Version:        v1.00a
-- Description:    Implement the functions needed for shift right and the
--     logical instructions
--
-------------------------------------------------------------------------------
-- Structure:
--     shift_logic.vhd
--
-------------------------------------------------------------------------------
-- Author:                 goran
-- History:
--     goran 2001-03-05    First Version of entity
--     goran 2001-03-09    First Version of architecture
--
-------------------------------------------------------------------------------
-- Naming Conventions:
--     active low signals:                  «*_n»
--     clock signals:                       «clk», «clk_div#», «clk_#x»
--     reset signals:                       «rst», «rst_n»
--     generics:                            «C_*»
--     user defined types:                  «*_TYPE»
--     state machine next state:            «*_ns»
--     state machine current state:         «*_cs»
--     combinatorial signals:               «*_com»
--     pipelined or register delay signals: «*_d#»
--     counter signals:                     «*cnt*»
--     clock enable signals:                «*_ce»
--     internal version of output port      «*_i»
--     device pins:                         «*_pin»
--     ports:                               - Names begin with Uppercase
--     processes:                           «*_PROCESS»
--     component instantiations:            «<ENTITY_>I_<#|FUNC>
-------------------------------------------------------------------------------
```

Пример 13. Заголовок файла проекта и соглашения о именах

Следующий вопрос — это соглашения об именах

Вопрос об именах для сигналов и переменных, так же как и все предыдущие темы, описанные в данном разделе, может показаться неважным только с первого взгляда. На самом деле, накопив некоторый опыт по разработке проектов на языках HDL, вы, уважаемый читатель, рано или поздно поймаете себя на мысли о том, что самые большие трудности в проекте вы испытали, придумывая имена для сигналов. Однообразие и унификация в проекте имен для сигналов — мощный инструмент, позволяющий вам сосредоточиться на том, что действительно написано в проекте, а не разгадывать шараду из многих и многих аббревиатур, которые вы нашли в проекте у соседа по комнате или в файле, взятом вами в архиве предыдущей темы. Унификация имен действительно сокращает время на разработку и действительно уменьшает число ошибок.

Сначала по форме. Существуют две формы написания имен:

- Первая — с применением символа подчеркивания, например,

```
это_первая_форма_написания_имени_сигнала.
```

- Вторая — с применением заглавных символов, намример,

```
ЭтоВтораяФормаНаписанияИмениСигнала.
```

Теперь по содержанию. Название сигнала должно отражать выполняемую им функцию. Нет необходимости экономить время на том, чтобы написать в имени сигнала несколько лишних букв. Помните, это с лихвой окупится на сопровождении проекта. В этой книге приведено достаточно примеров, в которых можно посмотреть систему названий сигналов. Далее еще будут приведены примеры, которые тоже могут быть изучены на предмет определения названий сигналов. Несколько забегая вперед, можно сказать следующее. При «печатании» файла проекта нет никакой необходимости каждую букву писать вручную. Существуют шаблоны типовых конструкций языка. Эти шаблоны встроены в программные инструменты, поставляемые фирмами-изготовителями. Кроме этого, существует достаточно много редакторов текста, которые позволяют пользователю создавать собственные шаблоны и вставлять их в текст, например EditPlus2. Так что всегда можно взять свой предыдущий проект, выделить из него «любимые» шаблоны и вставить их в качестве шаблонов редактора текста. Таким образом, двойной клик по строке с шаблоном добавит в ваш проект значительный кусок текста, причем уже с «любимыми» названиями проводов. Но даже если вы используете шаблоны, где названия проводов не написаны, то для того чтобы их заме-

нить, нужно только быстро научиться делать две процедуры — «выделить» и «заменить».

Далее, общепринято, что сигналы синхрочастоты называются **clock** или **clk**. Но довольно часто этого названия оказывается недостаточно. В тех проектах, где одновременно к кристаллу подводятся или из кристалла выводятся несколько частот, необходимо расширить данное имя. Например, к имени может быть добавлено в качестве расширения значение данной тактовой частоты, например: **clk50MHz** и **clk133MHz**. Или к имени синхрочастоты добавляется расширение имени, определяющее роль данной частоты в проекте. Например, для системной (основной) частоты: **sysclk**. Если в проекте есть несколько доменов синхрочастот, то расширение, указывающее, к какому именно домену синхрочастоты относится сигнал, может быть добавлено к названию сигнала, находящегося в данном домене. То есть это расширение может быть добавлено к сигналам данных, управления и к служебным сигналам.

Теперь необходимо остановиться на том, что одни сигналы будут активны при высоком уровне сигнала, а другие — при низком. Это тоже может быть отражено в имени сигнала. Варианты следующие.

- Название сигнала начинается с символа «/», например «**/Reset**».
- Название сигнала начинается или заканчивается символом «**N**», например «**NReset**» или «**reset_n**».
- В названии сигнала участвует символ «#».

Тоже самое относится и к сигналу **Reset**. В проектах довольно часто встречаются «**/Reset**», «**rst**», «**nrst**», «**rst_n**» и так далее.

Все сказанное выше об именах сигналов относилось, в первую очередь, к именам модулей нижнего уровня. Сложнее обстоит дело с модулями верхних уровней. Представим себе ситуацию, когда в проекте установлены несколько компонентов, например таких, как UART, или несколько компонентов, таких, как контроллер SPI. В этом случае каждый из установленных компонентов имеет свое уникальное имя в иерархии, но все эти компоненты имеют одинаковые имена портов. В такой ситуации сигналы, связывающие компоненты, должны в своем имени отражать не только функциональную сущность сигнала, но также указывать, от какого и до какого компонента данный сигнал передает информацию. В системе, разработанной для унификации имен проекта «шина Авалон», применяемой для связи с софт процессором Ниос, приняты следующие правила составления имен для сигналов.

Если сигнал от компонента выходит на внешний порт, то название такого сигнала на проекте верхнего уровня будет выглядеть так:

```
<promoted-port-name> = <port-name>_<to|from|to_and_from>_
<instance-name>,
```

где «<promoted-port-name>» является формальным hdl-названием порта ввода/вывода, модуля верхнего уровня Системы Авалона; <port-name> — формальное hdl-название порта на внутреннем модуле; <instance-name> — формальное hdl-название внутреннего модуля Системы Авалона. Также имена имеют в середине вставки «to»_ или «from» или «to_and_from», в зависимости от того как работает на модуле данный выход — это ввод, вывод или двунаправленный вывод.

Такая же система кодирования имен применена и для шин Системы Авалона. Для портов системного уровня кодировка выглядит так:

```
a) <system-port-name> = <avalon-role>_<to|from|to_and_from>_
<instance-name>,
```

где <avalon-role> — кодировка сигнала управления шиной, например сигнал чтения или записи.

Для общедоступных портов схема названий такова:

```
b) <system-port-name> = <tri-state-bus-group-name>_<avalon-role>.
```

В результате названия сигналов выглядят довольно странно, но тем не менее такая форма записи удобна для автоматической генерации и такие названия дают полную информацию о сигналах, их цели, назначении и роли в системе:

```
bidir_port_to_and_from_the_lcd_pio
txd_from_the_printf_uart
```

Можно порекомендовать читателям дополнительно ознакомиться с описаниями шин, применяемых для подключения ядер IP. Описания таких шин предлагаются фирмами — производителями ядер IP, а также эти описания можно найти на сайте opencores.org. Одним из примеров таких шин является шина WISHBONE, описанная в [2]. В описании на шину надо прочитать разделы соглашения об именах и об описании сигналов. Для разработки проектов, в которых будут применяться IP ядра, взятые с сайта opencores.org, применение данной шины будет целесообразным, так как это обеспечит единый интерфейс во всем проекте.

В заключение данного раздела мы обсудим комментарии в проектах. Что касается сопровождения проекта, то здесь было все обсуждено достаточно подробно. Тем не менее еще раз хочется напомнить, что в файле проекта совсем не лишними будут не только обычные комментарии, но и минимально необходимые справочные данные, описания основных сигналов или даже фрагменты диаграмм сигналов. В примере 14 показано размещение справочной информации в файле проекта. В примере 15 показано размещение небольшой диаграммы сигналов в файле проекта.

```
VHDL файл
rb8_node_tg: DFFE
PORT MAP ( d => Inp_shift_rg8bit, clk => sysclk, clrn => rb8_r_node,
ena => sbuf_rx_load_ena_node, q => rb8_node);-- M1

--=============================================
--Where SM0, SM1 specify the serial port mode as follows:
---------------------------------------------------------------
--SM0| SM1 | Mode| Description | Baud Rate
---------------------------------------------------------------
--0    | 0    | 0    | Shift          | Register fixed | (f OSC ./12)
--0    | 1    | 1    | 8-bit UART | variable  | (set by timer)
--1    | 0    | 2    | 9-bit UART | fixed     | (f OSC ./64 or f OSC ./32)
--1    | 1    | 3    | 9-bit UART | variable  | (set by timer)
---------------------------------------------------------------
Shift_mode <= (( NOT SM0) AND ( NOT SM1)) ;
Uart8_mode <= (( NOT SM0) AND SM1);
Uart9_mode <= SM0;
```

Пример 14. Размещение табличной справочной информации в файле проекта

```
/*
____/TTTTTT\____/XX..XX\___ шина данных
_____/TT\_____________ строб записи
      ^
                              запись в регистр
*/
```

Пример 15. Размещение графической справочной информации в файле проекта

На этом примеры выполнения файлов закончены. Теперь мы перейдем к шаблонам для файлов проекта, которые пользователь может найти в программных инструментах.

Литература

1. Голуб А. И. Веревка достаточной длины, чтобы… выстрелить себе в ногу: Правила программирования на Си и Си++. Москва. 2001.
2. WISHBONE SoC Architecture Specification, Revision B.3; **http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf**