Продолжение. Начало в № 3`2008

Иосиф КАРШЕНБОЙМ
iosifk@.narod.ru

Краткий курс HDL.
Часть 2. Описание языка

Системные функции $fopen, $fdisplay, $fstrobe, $fwrite, $ftell, $feof, $ferror, $fgetc, $fgets и $fclose

Команда открытия файла

Команда $fopen используется для того, чтобы открыть файл для чтения, записи и/или добавления данных. Эта операция должна предшествовать любой команде чтения или записи, определенной в этом документе. Используя $fopen, можно определить имя файла и режим работы с данным файлом — чтение, запись и т. д. Синтаксис команды: $fopen («<file_name>», «<file_mode>»).

После открытия файла будет установлен номер handle для данного файла, и его необходимо использовать в последующих командах при обращении к данному файлу. Для этого должна быть объявлена переменная, имеющая размерность целого числа, и на эту переменную назначается значение хендлера.

Режим открытия файла:

- «r» — файл ASCII для чтения;
- «rb» — двоичный файл для читения;
- «w» — файл ASCII для записи (если файл уже существует, то его содержимое удаляется);
- «wb» — двоичный файл для записи (если файл уже существует, то его содержимое удаляется);
- «a» — файл ASCII для записи (если файл уже существует, то к концу этого файла добавляются новые данные);
- «ab» — двоичный файл для записи (если файл уже существует, то к концу этого файла добавляются новые данные);
- «r +» — файл ASCII для чтения и записи.

Команда записи

Команда $fdisplay производит запись данных в указанном файле в соответствии с выбранным форматом. При использовании этой функции в файл могут быть записаны данные из текста в определенном формате, а также из системных функций, задач и значений сигнала. При выдаче команды $fopen должен быть назначен хендлер файла, который и будет указывать на то, куда будет производиться запись. Синтаксис команды:

```
$fdisplay (<file_desc>, «<string>», variables);
```

Команда $fwrite работает аналогично команде $fdisplay, которая тоже может записать указанную строку в файл, но только с той разницей, что она не выполняет перевод каретки после выполнения операции.

Команда $fstrobe также работает аналогично команде $fdisplay, но только перед записью сообщения она ждет, чтобы произошли все события моделирования, находящиеся в очереди.

Команда $fmonitor произведет запись строки в требуемый файл всякий раз, когда будет обнаружено изменение в значении для одной из переменных. После того как строка будет записана, выполняется перевод каретки.

При помощи команд записи ($fdisplay, $fwrite, $fstrobe, $fmonitor) переменные могут быть записаны в одном из требуемых форматов. Для того чтобы определить специальные символы или форматирование, можно использовать специальные ESC-символы. Эти форматы перечислены ниже.

Переменные:

- %b — бинарное значение;
- %h — шестнадцатеричное значение;
- %d — десятичное значение;
- %t — время;
- %s — строка;
- %c — ASCII;
- %f — реальное значение;
- %e — экспоненциальное значение;
- %o — восьмеричное значение;
- %m — иерархическое название модуля;
- %v — сила (Strength).

Символы ESC:

- \t — табуляция (Tab);
- \n — переход на новую линию;
- \\ — символ «наклонная черта влево» (Backslash);
- %% — символ «процент»;
- \» — символ «кавычки»;
- \<octal> — восьмеричное представление ASCII.

Команды чтения

Команда $fgets будет читать всю строку текста из файла и хранить эту информацию как строку.

Формат для $fgets:

```
$fgets (<string_reg>, <file_desc>);
```

Команда $fgets возвращает целочисленное значение или указание чтения числа символов, или нулевую индикацию ошибки при попытке чтения. В параметре <string_reg> должна быть определена разрядность для читаемой строки, равная числу символов в самой длинной строке, умноженной на 8.

Команда $fgetc будет читать символ из файла и возвращать его как 8-битовую строку. Попытка чтения из конца файла (EOF), приводит к выдаче значения «–1». Команда $fscanf будет читать строку из файла и хранить ее в указанной форме.

Формат для $fscanf:

```
$fscanf (<file_desc>, <format>, <destination_regs>);
```

где формат команды аналогичен тому, как определено в команде чтения выше и, кроме этого, параметр <destination_regs> указывает на то, где хранятся данные, полученные при чтении. Команда $fscanf возвратит целочисленное значение, указывающее число прочитанных данных. Если в течение операции чтения произойдет ошибка, то это число будет нулем.

Специальные функции

Команда $ferror проверяет последнюю ошибку, которая произошла при чтении или записи из открытого файла, и выдает об этом сообщение. Строка данных при записи может иметь длину до 80 символов (640 битов).

Команда $fseek снова установит указатель в открытом файле в указанную позицию. Формат для $fseek команды:

```
$fseek (<file_desc>, <offset_value>, <operation_number>);
```

где <operation_number> может иметь одно из трех значений: 0 — установить указатель на начало файла и это положение использовать при чтении данных; 1 — установить указатель на текущее положение и это положение использовать при чтении данных; 2 — установить указатель на конец файла и это положение использовать при чтении данных.

Команда $fseek возвратит нуль, если команда была успешна, и «–1», если нет.

Команда $ftell определяет позицию указателя в файле, выдавая целочисленное значение, указывающее число байтов смещения от начала файла.

Команда $fflush пишет любые буферизированные данные в указанный файл.

Команда закрытия файла

Команда $fclose закрывает предыдущий открытый файл. Формат $fclose:

```
$fclose (<file_desc>);
```

Желательно ограничить количество и объем операций чтения и записи в файлы при моделировании, поскольку это может значительно увеличить полное время выполнения моделирования. Доступ к файлу может быть довольно медленным процессом, и если его выполнять часто, то это может увеличить затраты времени на моделирование.

Далее в примерах 28, 29 приводятся коды, необходимые для записи и чтения состояний сигналов в файлы.

```
// ------------------------------------
// Example of writing monitored signals:
// ------------------------------------

   // Define file handle integer
   integer outfile;

   initial begin
       // Open file output.dat for writing
       outfile = $fopen(outfile, «output.dat», «w»);

   // Check if file was properly opened and if not, produce error and exit
       if (outfile == 0) begin
           $display(«Error: File, output.dat could not be opened.\
nExiting Simulation.»);
           $finish;
       end

       // Write monitor data to a file
       $fmonitor (outfile, «Time: %t\t Data_out = %h», $realtime,
Data_out);
       // Wait for 1 ms and end monitoring
       #1000000;

       // Close file to end monitoring
       $fclose(outfile);
   end
```

Пример 28. Коды, необходимые для записи состояний сигналов в файл

```
// Example of reading a file using $fscanf:
// ----------------------------------------

   real number;

   // Define integers for file handling
   integer number_file;
   integer i=1;

   initial begin
       // Open file numbers.txt for reading
       number_file = $fopen(«numbers.txt», «r»);
       // Produce error and exit if file could not be opened
       if (number_file == 0) begin
           $display(«Error: Failed to open file, numbers.txt\nExiting
Simulation.»);
           $finish;
       end
       // Loop while data is being read from file
       // (i will be -1 when end of file or 0 for blank line)
       while (i>0) begin
           $display(«i = %d», i);
           i=$fscanf(number_file, «%f», number);
           $display(«Number read from file is %f», number);
           @(posedge CLK);
       end
       // Close out file when finished reading
       $fclose(number_file);
       #100;
       $display(«Simulation ended normally»);
       $stop;
   end
```

Пример 29. Коды, необходимые для чтения состояний сигналов из файла

Директивы компилятора 'ifdef', `ifndef, `else, `elsif, `endif и `define

Условная компиляция — 'ifdef', `ifndef, `else, `elsif и `endif

Условная компиляция проекта может быть выполнена при использовании директив компилятора 'ifdef', `ifndef, `else, elsif, и `endif. Для описания директив компилятора используется значок «`» апострофа. Директивы условной компиляции `ifdef и `ifndef могут находиться в любом месте проекта, они применяются для того, чтобы по требуемому условию выполнить компиляцию необходимых утверждений, модулей, блоков, объявлений и других директив компилятора. Директива `else является дополнительной, и только эта одна директива используется после `ifdef или `ifndef. Любое число директив `elsif может сопровождать директивы `ifdef или `ifndef. Но они всегда заканчиваются директивой `endif. Приведем пример условной компиляции (примеры 30, 31).

```
//Следующий фрагмент текста выполнится,
// если TEST определено
'ifdef TEST
 module test;
...
...
endmodule
// Этот фрагмент выполнится, по умолчанию,
// если TEST не определено
'else
...
...
endmodule
'endif
```

Пример 30. Пример условной компиляции

```
module top;
   ... — выполняются безусловно
'ifdef A1 — выполняется, если A1 определено
   ...

'elsif A2 — выполняется, если A2 определено
   ...
'else — выполняется, по умолчанию
   ...
'endif
'ifndef A3 — выполняется, если A3 не определено
   ...
'endif
endmodule
```

Пример 31. Пример условной компиляции

Директива компилятора `define

Директива компилятора `define используется так же, как и в других языках программирования. Ее применение для описания текстового макроса можно представить следующим образом. Сначала выполняется определение макроса:

```
`define my_clock @(posedge clk);
```

А в коде программы делается так:

```
always `my_clock;
```

или

```
a<= `my_clock b;
```

Теперь, когда подготовительная часть описания пройдена, переходим к следующему разделу книги. Можно сказать, что мы выучили слова, теперь научимся складывать их во фразы.

В своей книге [2, 3] Жаник Бергерон делает вот такие, очень важные для нас выводы: до 70% трудозатрат в проекте приходится на верификацию проекта. В дальнейшем мы еще вернемся к рассмотрению данного вопроса. А сейчас давайте обратимся к тому, как именно производится разработка проекта и какое место в ней занимает язык Verilog.

Этапы ведения проекта

Типовой проект ведется следующим образом (рис. 1). Проект начинается с проработки технического задания на проектирование и создания спецификаций проекта. Далее выполняется поведенческое описание проекта. На этом этапе моделируется то, как проект будет взаимодействовать с внешней средой. После того как этот этап выполнен, разработчики переходят к одному из наиболее существенных этапов — RTL-описанию. На этом этапе производится моделирование работы устройства на уровне передачи информации между регистрами.

Далее производят компиляцию проекта под конкретную технологию, проверяют временные соотношения сигналов и производят окончательную трассировку на физическом уровне.

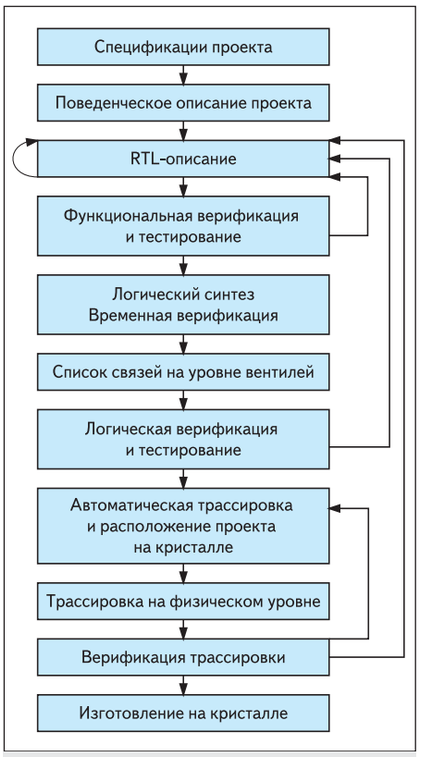

Рис. 1. Этапы ведения проекта

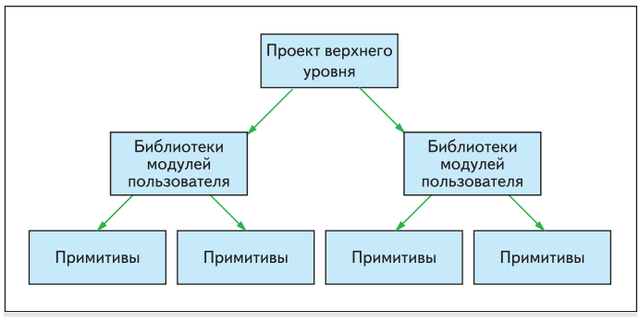

Рис. 2. Нисходящий процесс проектирования

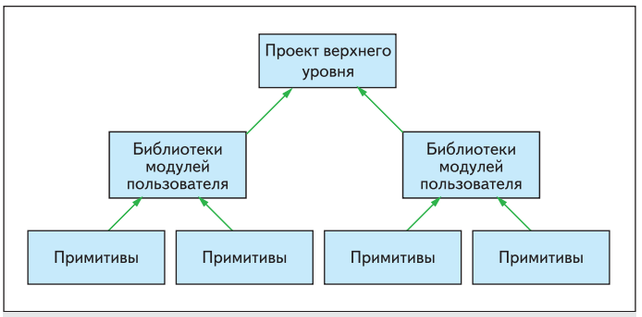

Рис. 3. Восходящий процесс проектирования

Преимущества использования языков HDL

Проектирование на языке HDL имеет много преимуществ по сравнению с традиционным проектом на основе схемного описания:

- Проекты могут быть выполнены на очень абстрактном уровне при помощи HDL. Проектировщики могут выполнить описание на уровне RTL, не выбирая определенную технологию изготовления. Логические инструментальные средства синтеза могут автоматически преобразовать проект в любую технологию изготовления. Если появляется новая технология, то специалисты не должны перепроектировать всю схему целиком. Они просто вводят в логический инструмент синтеза новое описание RTL-уровня и создают новый netlist, в соответствии с новой технологией изготовления микросхем. Логический инструмент синтеза оптимизирует схему для новой технологии по занимаемой на кристалле площади и по временным задержкам.
- Если проекты пользователя выполнены на языке HDL, то функциональная проверка проекта может быть сделана непосредственно в цикле проекта, на ранней стадии. Выполняя проверку проекта уже на уровне RTL, проектировщики могут оптимизировать и изменять описание RTL до тех пор, пока оно не начнет выполнять требуемые для проекта функции. Большинство ошибок в проектах устраняется именно на этом этапе. Это значительно сокращает время проектирования, потому что при устранении ошибок нет необходимости многократно выполнять достаточно большой объем работ по размещению проекта на кристалле на уровне вентилей.
- Проектирование на языке HDL очень похоже на программирование. Текстовое описание с комментариями — это более простой способ разработки и отладки схемы. Текстовое описание выглядит более компактным, по сравнению со схемными решениями, представленными на уровне примитивов — триггеров и вентилей. Даже для проектов средней сложности, схемы, выполненные на уровне триггеров и вентилей, довольно трудны в сопровождении и дальнейшей модернизации.

Методологии ведения проекта

Есть две основных методики ведения проектов: нисходящая методология проектирования, когда проект ведется сверху вниз, и восходящая методология проектирования, когда проект ведется снизу вверх. В нисходящей методологии проектирования сначала определяется блок верхнего уровня и, исходя из заложенных в него функций, производится определение субблоков, которые будут необходимы для того, чтобы выполнять функции для блока верхнего уровня. Далее идет процесс разбивки субблоков на более мелкие блоки до тех пор, пока не будет достигнут уровень базовых ячеек, таких, которые не могут быть разделены на более мелкие ячейки. На рис. 2 показан нисходящий процесс проектирования.

В восходящей методологии проектирования сначала определяются стандартные блоки, которые могут быть использованы в проекте. Это могут быть как библиотечные блоки, поставляемые производителями микросхем, так и блоки, разработанные ранее. Используя эти стандартные блоки, строят большие ячейки, которые, в свою очередь, используют для более высокоуровневых блоков. Далее процесс строится аналогично до тех пор, пока не будет сформирован блок самого верхнего уровня в данном проекте. На рис. 3 показан восходящий процесс проектирования.

Как правило, в проекте используется комбинация нисходящего и восходящего потоков проектирования. Ведущие архитекторы дизайна определяют спецификации блока верхнего уровня. Разработчики проекта решают то, как проект должен быть выполнен, разбивая его на функциональные узлы и выделяя такие узлы в отдельные блоки и субблоки. В то же самое время разработчики могут выполнять схемы нижнего уровня, оптимизированные для конкретного приложения, например под конкретную серию микросхем определенного производителя. Из этих ячеек нижнего уровня строятся более высокоуровневые ячейки. Обе части проекта — восходящая и нисходящая — встречаются вместе, и, таким образом, появляется возможность оптимально произвести разбивку проекта на отдельные блоки и субблоки.

О разделении проекта на части в FPGA

Еще несколько слов о разделении проекта на части. Если вы будете читать учебник по языку Verilog, написанный иностранным автором, то, несомненно, обратите внимание на то, что такой учебник будет предназначен, скорее всего, для разработчиков микросхем. Однако для отечественного читателя гораздо больший интерес представляет учебник, ориентированный на пользователей FPGA. Так вот, при работе с FPGA необходимо обратить внимание на то, что некоторые библиотечные блоки и примитивы, поставляемые производителями микросхем, описывают аппаратные ресурсы модулей. К таким ресурсам относятся буферы ввода/вывода, буферы вывода с третьим состоянием, блоки управления частотой и т. д. Эти блоки должны быть расположены только на верхнем уровне проекта. То же самое относится и к двунаправленным шинам с третьим состоянием. Если внутренняя организация микросхемы FPGA не поддерживает шины с третьим состоянием, то такие шины могут располагаться только на верхнем уровне проекта. Таким образом, один уровень иерархии образуется «автоматически». И проект пользователя будет теперь состоять, как минимум, из двух частей: логического ядра проекта пользователя и оболочки, которая «привязывает данное ядро» к кристаллу FPGA. При таком построении проекта разработчик не испытывает трудностей при симуляции проекта на RTL-уровне. Все сигналы в логическом ядре проекта пользователя будут только одного логического уровня, и все они будут только однонаправленные. Дальнейшее рассмотрение разделения проекта на части будет продолжено в разделе о метастабильности и о разделении проекта на различные клоковые домены.

Модули (Modules)

Как определяются понятия блоков и субблоков при иерархическом моделировании на Verilog? Язык Verilog имеет термин **модуль**. Модуль — это основной стандартный блок при описании на языке Verilog. Модуль может быть как элементом библиотеки самого нижнего уровня, так и блоком, содержащим

несколько субблоков низшего уровня. Как правило, субблоки группируются в модули для того, чтобы обеспечить общие функциональные возможности, которые многократно используются во многих местах проекта. Если проводить аналогию с Си++, то понятие **модуль** в Verilog можно соотнести с понятием **класс** в Си++.

Если рассматривать с точки зрения проекта верхнего уровня, то модуль обеспечивает требуемые функциональные возможности через интерфейс портов (вводы и выводы), но, при этом, вся «начинка» модуля и ее функционирование будет скрыто для проекта верхнего уровня. Это позволяет проектировщику гибко менять внутреннюю организацию модуля, не затрагивая остальную часть дизайна. Если провести аналогию со схемным описанием проекта, то схема тоже может быть выполнена как иерархическая структура. На верхнем уровне иерархии схема представляет собой набор квадратиков, представляющих схемы более низкого уровня, и провода, связывающие эти квадратики. Каждый модуль нижнего уровня имеет место для подключения проводов — разъем или контакт на схеме. В Verilog'е такое место, аналогичное разъему на схеме, называется **порт**.

В примере 32 показано описание модуля. В языке Verilog модуль объявляется при помощи ключевого слова **module**. Соответствующее ключевое слово **endmodule** должно быть написано в конце определения модуля. Каждый модуль должен иметь имя — **module_name**, которое является идентификатором для модуля, и список входов и выходов модуля — **module_terminal_list**.

```
module <module_name> (<module_terminal_list>);
...
<module internals>
...
...
endmodule
```

Пример 32. Шаблон описания модуля

Например, триггер типа T (T-flipflop) может быть определен как модуль следующим образом (пример 33).

```
module T_FF (q, clock, reset);
.
.
<functionality of T-flipflop>
.
.
endmodule
```

Пример 33. Коды, необходимые для определения триггера типа T (T-flipflop) как модуля

Теперь рассмотрим более подробно модули и порты с точки зрения языка Verilog

Итак, как было сказано выше, модуль — это основной стандартный блок при моделировании иерархических проектов.

До сих пор не рассматривалась внутренняя организация модулей, и описывалось только то, как модули определены и инстанциированы. В этом разделе мы анализируем внутреннюю организацию модуля более детально.

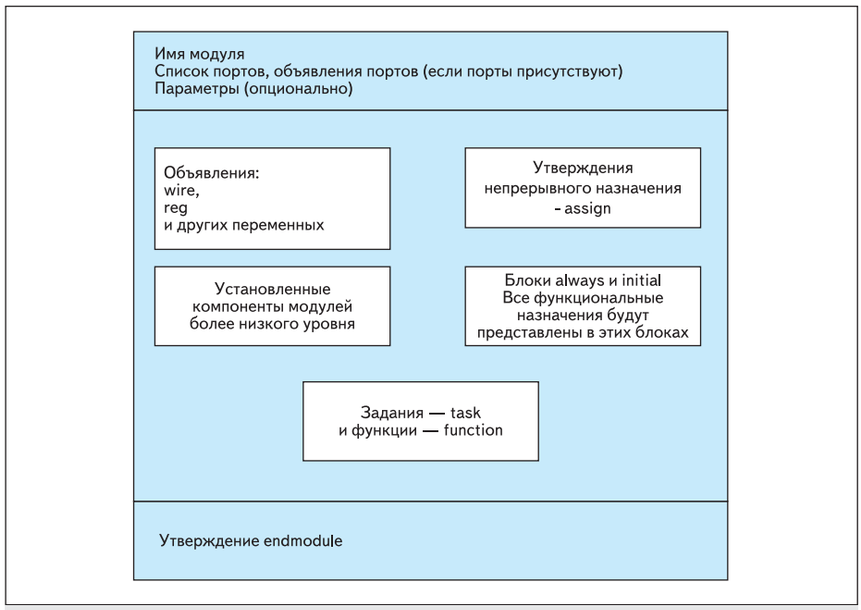

Рис. 4. Компоненты модуля в языке Verilog

Модуль в Verilog состоит из нескольких частей, так, как показано на рис. 4.

Определение модуля всегда начинается с ключевого слова **module**. Название модуля, список портов модуля, объявления портов и дополнительные параметры должны быть выбраны в начале определения модуля. Список портов и объявления портов имеют место только в том случае, если модуль имеет какие-либо порты. Порты предназначены для того, чтобы взаимодействовать с внешними сигналами или другими модулями.

У модуля есть пять компонентов:

- объявления переменных (variable declarations);
- операторы прохождения данных (dataflow statements);
- установленные компоненты модулей (инстансы) более низкого уровня (instantiation of lower modules);
- блоки с поведенческим описанием (behavioral blocks)
- и задачи или функций (tasks or functions).

Эти компоненты могут быть расположены в любом порядке и в любом месте в описании модуля. Слово **endmodule** должно всегда стоять в конце определения модуля. Все компоненты, кроме ключевого слова — **module**, названия модуля и **endmodule**, являются дополнительными и могут быть расположены в любом месте в описании модуля, в соответствии с тем, как это нужно разработчику при описании модуля. Verilog позволяет выполнить несколько описаний модулей в одном файле. В таком файле модули могут быть расположены в любом порядке.

Модуль обеспечивает шаблон, из которого можно создать фактические компоненты, устанавливаемые в проект. Компилятор производит вызов файла модуля, и создается новый объект — уникальный компонент из шаблона модуля. Каждый объект имеет свое собственное уникальное имя, свои переменные, параметры и интерфейс ввода/вывода. Процесс создания объектов от шаблона модуля называют установкой (instantiation) компонента, и сами объекты называют компонентами (instance). В примере 34 в блоке верхнего уровня создаются четыре компонента из одного шаблона T-flipflop (T_FF). Каждый компонент T_FF содержит триггер D_FF и инвертирующий вентиль. Каждому из этих четырех компонентов дано уникальное имя.

```
// Define the top-level module called ripple carry
// counter. It instantiates 4 T-flipflops. Interconnections are
// shown in Section 2.2, 4-bit Ripple Carry Counter.
module ripple_carry_counter(q, clk, reset);
   output [3:0] q; //I/O signals and vector declarations
   //will be explained later.
   input clk, reset; //I/O signals will be explained later.
   //Four instances of the module T_FF are created. Each has a unique
   //name.Each instance is passed a set of signals. Notice, that
   //each instance is a copy of the module T_FF.
   T_FF tff0(q[0],clk, reset);
   T_FF tff1(q[1],q[0], reset);
   T_FF tff2(q[2],q[1], reset);
   T_FF tff3(q[3],q[2], reset);
endmodule
// Define the module T_FF. It instantiates a D-flipflop. We assumed
// that module D-flipflop is defined elsewhere in the design. Refer
// to Figure 2-4 for interconnections.
module T_FF(q, clk, reset);
   //Declarations to be explained later
   output q;
   input clk, reset;
   wire d;
   D_FF dff0(q, d, clk, reset); // Instantiate D_FF. Call it dff0.
   not n1(d, q); // not gate is a Verilog primitive. Explained later.
endmodule
```

Пример 34. Установка компонентов в проекте

В языке Verilog нет возможности делать вложенные описания модулей. Одно определение модуля не может содержать другое определение модуля в пределах определений

module и **endmodule**. Вместо этого описание одного модуля может включать установленные компоненты от других модулей. Важно не путать описание модуля и установленные компоненты — копии этого модуля. Определение модуля просто описывает то, как модуль будет работать, его внутреннюю организацию и его интерфейс.

Для того чтобы использовать модуль в проекте, необходимо выполнить установку компонента — копии этого модуля и в самом начале файла необходимо вставить директиву препроцессора `include с именем того файла, в котором находится описание устанавливаемого модуля.

Синтаксис: `include **имя_файла**. Например, `include ripple carry.v.

Макромодули

Применение макромодуля делает моделирование более эффективным, при этом выполняется определение макромодуля в соответствии с определением вызывающего (родительского) модуля. Однако компилятор HDL обрабатывает конструкцию макромодуля так же, как и конструкцию модуля. Используете ли вы модуль или макромодуль, в процессе синтеза будет создана та же самая иерархическая конструкция. На примере 35 показано то, как использовать конструкцию макромодуля.

```
macromodule adder (in1,in2,out1);
input [3:0] in1,in2;
output [4:0] out1;
assign out1 = in1 + in2;
endmodule
```

Пример 35. Использование конструкции макромодуля

Порты

Порты обеспечивают интерфейс, с помощью которого модуль может общаться с внешней средой. Например, входы ввода/вывода сигналов из микросхемы представляют собой ее порты. Среда может взаимодействовать с модулем только через порты модуля. Внутренняя организация модуля недоступна извне. Это обеспечивает проектировщику очень большие возможности, потому что внутренняя организация модуля может быть изменена, но это не затрагивает внешнюю, по отношению к модулю, среду до тех пор, пока интерфейс модуля не будет изменен.

Порты модуля определяются следующим образом: после имени модуля в круглых скобках приводится список портов модуля так, как показано здесь:

```
module name ( port_list );
```

Список портов

Определение модуля содержит список портов. Если модуль не обменивается сигналами со средой, то у него нет списка портов.

Порт в списке может быть выражен следующим образом:

- идентификатор;
- единственный бит, выбранный из вектора, объявленного в пределах модуля;
- группа битов, выбранных из вектора, объявленного в пределах модуля;
- конкатенация любого из вышеупомянутых (конкатенация — процесс объединения нескольких однобитовых или многобитовых операндов в один большой битовый вектор. Для получения дополнительной информации см. раздел «Оператор конкатенации»).

Каждый порт в списке должен быть объявлен как ввод, вывод или двунаправленный порт при помощи ключевых слов **input**, **output** или **inout**.

Рассмотрим 4-битовый полный сумматор, который установлен в проекте верхнего уровня. Диаграмма для портов ввода/вывода показана на рис. 5.

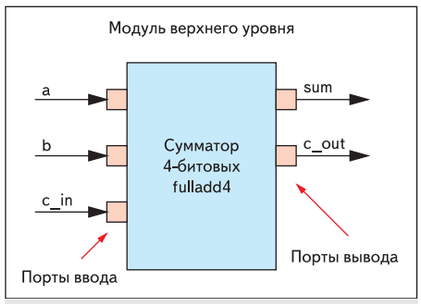

Рис. 5. Порты ввода/вывода для полного сумматора, установленного в проекте верхнего уровня (у модуля верхнего уровня нет собственных портов)

Обратите внимание на то, что проект **Top** — это проект верхнего уровня. На модуль fulladd4 входные сигналы поступают на порты **a**, **b** и **c_in**, а выходные сигналы, формируемые модулем, выдаются из него через порты **sum** и **c_out**. Модуль **Top** — модуль верхнего уровня в моделировании, и к нему нет необходимости передавать сигналы или получать их извне. Таким образом, этот модуль не имеет списка портов. Названия модуля и списки портов для обоих объявлений модулей в Verilog приведены в примере 36.

```
module fulladd4(sum, c_out, a, b, c_in); //Module with a list of ports
module Top; // No list of ports, top-level module in simulation
```

Пример 36. Список портов для модулей Top и fulladd4

Стили описания — функциональное (поведенческое) и структурное

Verilog является языком, который позволяет выполнять как функциональное или поведенческое, так и структурное описание.

Структурные описания

Элементами структурного описания в языке Verilog являются различные логические вентили, специфические библиотечные компоненты и компоненты, определяемые пользователем. Само структурное описание как таковое представляет собой набор компонентов, установленных в проект и связанных проводами. В простейшем случае структурное описание может рассматриваться как простой netlist, в котором представлены установленные в проекте вентили, порты которых связаны проводами между собой. Однако, в отличие от списка связей (netlist), цепи в структурном описании могут быть также представлены в виде произвольных выражений, которые описывают то, как будет функционировать та или иная цепь. Такое назначение на цепь называют непрерывным назначением. Непрерывные назначения — удобное средство для связи между простыми списками связей (netlist) и функциональными описаниями.

Структурное описание Verilog может содержать различные иерархические конструкции и конструкции уровня вентилей, а также определения модулей, установленные компоненты и подключения из netlist.

Функциональное описание

Функциональные элементы описания в языке Verilog — это, в основном, объявления функций, задачи — **tasks** и блоки — **always**. Эти элементы описывают функцию схемы, именно то, как она работает, но не описывают ее физическую сущность или размещение на кристалле. Выбор вентилей и компонентов оставляют полностью на усмотрение того программного инструмента, который будет проводить компиляцию проекта.

Разработчик может создать описания с функциональными конструкциями Verilog. Эти конструкции могут появиться в пределах функций или блоков (**always**). Функции подразумевают только комбинационную логику, а блоки (**always**) могут подразумевать как комбинационную логику, так и триггеры, регистры или счетчики.

Назначения (Assignments)

Краткий обзор назначений

Назначение — основной механизм для того, чтобы получить значения для цепи и для регистра. Есть две канонических формы назначения:

- непрерывное назначение, которое определяет значения на цепь;
- процедурное назначение, которое определяет значения на регистры.

Назначение состоит из двух частей, которые отделены символом равенства (=). Правая сторона может быть любым выражением, которое вычисляется до некоторого определенного значения. Левая сторона указывает переменную, на которую должно быть сделано назначение, вычисленное в соответствии с тем, что записано на правой стороне. Левая сторона может принять одну из следующих форм в зависимости от того, является ли на-

Таблица 19. Разрешенные формы назначений для левой стороны

Тип назначения	Выражение на левой стороне
Непрерывное назначение	• цепь (net)(vector или scalar) • определенный бит, выбранный из векторной цепи (constant bit select of a vector net) • определенная группа битов, выбранных из векторной цепи (constant part select of a vector net) • комбинация любых вариантов из трех перечисленных
Процедурное назначение	• регистр (register) (vector или scalar) • определенный бит, выбранный из регистра (bit select of a vector register) • определенная группа битов, выбранных из регистра (constant part select of a vector register) • элемент памяти (memory element) • комбинация любых вариантов из четырех перечисленных

значение непрерывным (**continuous assignment**) или процедурным (**procedural assignment**) (табл. 19).

Непрерывные назначения

Непрерывные назначения управляют значениями цепи, вектора и скаляра. Значение слова «непрерывное» — в том, что назначение происходит всякий раз, когда моделирование заставляет значение правой стороны изменяться. Непрерывные назначения обеспечивают способ моделировать комбинационную логику, не определяя взаимосвязь вентилей логики. Вместо этого модель определяет логическое выражение, которое управляет цепью. Выражение на правой стороне непрерывного назначения ничем не ограничено. Оно может даже содержать ссылку к функции. Таким образом, результат оператора выбора — **case**, условного оператора — **if** или другой процедурной конструкции может управлять цепью.

Пример синтаксиса непрерывного назначения:

```
<net_declaration>
    ||= trireg <charge_strength>? <expandrange>? <delay>?
<list_of_variables> ;
    ||= <NETTYPE> <drive_strength>? <expandrange>? <delay>?
        <list_of_assignments> ;
<continuous_assign>
    ::= assign <drive_strength>? <delay>? <list_of_assignments> ;
<expandrange>
    ::= <range>
    ||= scalared <range>
    ||= vectored <range>
<range>
    ::= [ <constant_expression> : <constant_expression> ]
<list_of_assignments>
    ::= <assignment> <,<assignment>>*
<charge_strength>
    ::= ( small )
    ||= ( medium )
    ||= ( large )
<drive_strength>
    ::= ( <STRENGTH0> , <STRENGTH1> )
||= ( <STRENGTH1> , <STRENGTH0> )
```

Пример 37. Синтаксис для <net_declaration>

Назначение, выполняемое при объявлении цепи

Первые две альтернативы в <net_declaration> обсуждались в главе 3 «Типы данных» (см. раздел 3.2.3 синтаксиса объявления). Третья альтернатива (сетевые назначения), выполняемого при объявлении цепи, позволяет поместить непрерывное назначение на цепь в том же самом утверждении, которое объявляет эту цепь. Следующий пример для непрерывного назначения при <net_declaration>:

```
wire (strong1, pull0) mynet = enable;
```

Примечание. *Поскольку цепь может быть объявлена только однажды, только одно назначение на эту цепь может быть сделано при объявлении именно этой конкретной цепи. Это отличается от обычного непрерывного назначения, так как обычно одна цепь может получить многократные непрерывные назначения.*

Оператор непрерывного назначения — assign

Утверждение <continuous_assign> помещает непрерывное назначение на цепь, которая была предварительно объявлена, или явно, в соответствии с объявлением цепи, или неявно при использовании ее названия в списке цепей для примитива — вентиля, определяемого пользователем примитива, или устанавливаемого в проект экземпляра модуля. Вот пример непрерывного назначения на цепь, которая была предварительно объявлена:

```
assign (strong1, pull0) mynet = enable;
```

Назначения на цепях являются непрерывными и автоматическими. Это означает, что всякий раз, когда операнд, находящийся с правой стороны выражения, меняет свое значение в течение моделирования, то все, что находится с правой стороны выражения, пересчитывается и назначается на левую сторону выражения.

Вот пример использования непрерывного назначения для того, чтобы промоделировать сумматор на четыре бита с переносом (пример 38). Отметьте, что назначение не могло быть определено непосредственно в объявлении цепей, потому что оно требует конкатенации в левой части выражения.

```
module adder (sum_out, carry_out, carry_in, ina, inb);
output [3:0]sum_out;
input [3:0]ina, inb;
output carry_out;
input carry_in;
wire carry_out, carry_in;
wire[3:0] sum_out, ina, inb;
assign
  {carry_out, sum_out} = ina + inb + carry_in;
endmodule
```

Пример 38. Использование утверждения для непрерывного назначения

В примере 39 описан модуль с одной 16-разрядной выходной шиной. В нем производится выбор между одной из четырех входных шин, и выбранная шина подключается к выходной шине.

```
module select_bus (busout, bus0, bus1, bus2, bus3, enable, s);
parameter n=16;
parameter Zee=16'bz;
output [1:n ] busout;
input [1:n] bus0, bus1, bus2, bus3;
input enable;
input [1:2] s;
tri [1:n] data; // net declaration.
tri [1:n] busou t= enable ? data : Zee;   // net declaration with
                                          // continuous assignment.
assign // assignment statement with
   data = (s==0) ? bus0 : Zee,            // 4 continuous assignments.
   data = (s==1) ? bus1 : Zee,
   data = (s==2) ? bus2 : Zee,
   data = (s==3) ? bus3 : Zee;
endmodule
```

Пример 39. Назначение на цепь при объявлении и утверждении непрерывного назначения

В примере 39 приводится следующая последовательность событий при моделировании описания:

1. Имеется входная переменная **s**, значение которой проверяется при назначении цепи. И в соответствии с тем, какое значение принимает **s**, на шину **data** коммутируются данные от одной из четырех входных шин.
2. Для выходной шины при установке данных сделано непрерывное назначение в объявлении цепи: если установлен сигнал разрешения — **enable**, то содержание данных назначено на выходную шину, если же сигнал разрешения не принимает ни одного из значений, использованных для сравнения (0, 1, 2, 3, 4), то на выходную шину будет назначено значение **Zee**.

Задержки (delays)

Задержка (delay), заданная при непрерывном назначении, определяет продолжительность времени между изменением значения, находящегося справа от операнда, и назначением, сделанным на левую сторону операнда. Если то, что находится слева от операнда, представляет собой скалярную цепь, то задержка будет представлять собой такую же задерку, какая имеет место в примитивах — вентилях. То есть производится задержка того сигнала, который приходит на вентиль, причем можно дать различные задержки для нарастающего фронта сигнала, спадающего фронта сигнала и перехода сигнала в высокий импеданс.

Если то, что находится слева от операнда, представляет собой векторную цепь, то можно применить до трех различных задержек. Следующие правила определяют, какая из задержек будет выполняться при назначении:

- Если правая сторона была ненулевой и становится нулем, то используется задержка для спадающего фронта сигнала.
- Если правая сторона становится **z**, то используется задержка перехода сигнала в высокий импеданс.
- Для остальных случаев используется задержка для нарастающего фронта сигнала.

Если производится объявление цепи, то непосредственно в этом же объявлении можно

назначить и задержки на эту цепь. Такое назначение задержки отличается от того случая, который имеет место в объявлении задержки при непрерывном назначении на цепь. Значение задержки может быть применено в объявлении цепи так:

```
wire #10 wireA;
```

Эту форму записи применяют для того, чтобы показать, что в цепи, называемой **wireA**, имеется задержка в течение десяти единиц времени, прежде чем сигнал, передаваемый по цепи **wireA**, пройдет через эту цепь и поступит для моделирования во все утверждения, где участвует данный сигнал. А в том случае, когда задержка делается при непрерывном назначении в объявлении, она представляет собой только часть непрерывного назначения и не является задержкой всей цепи. Тогда она не будет добавлена к задержке других драйверов на этой цепи. Более того, эта задержка не добавляется, если производится назначение, приводящее к расширению векторной цепи (то есть добавляются цепи, которые не входили в первоначальное определение с ключевым словом **vector**); задержки на положительный фронт и задержки на отрицательный фронт не будут применены к индивидуальным битам, если назначение включено в объявления этих цепей.

Представим ситуацию, когда цепь имеет задержку. В этом случае, если операнд, находящийся с правой стороны, изменяется прежде, чем прошло время, необходимое для прохождения сигнала через эту цепь от предыдущего изменения, то тогда будет обработано лишь одно последнее изменение значения. И, таким образом, произойдет только одно назначение на цепь. Этот эффект называется инерционной задержкой.

В примере 40 осуществлена обработка вектора по исключающему ИЛИ. Разрядность шин и задержка определены как параметры, они могут быть изменены, когда экземпляры таких компонентов будут устанавливаться в проекты пользователя.

Также здесь необходимо добавить, что назначение задержек актуально только для симуляции проекта. Если разработчику нужна лишь RTL-модель, то назначение задержек можно не делать. Если же речь идет о модели, полученной после размещения проекта на кристалле, то в такой модели используются задержки, полученные в результате работы компилятора и трассировщика.

```
module modxor (axorb, a, b);
parameter size=8, delay=15;
output [size-1:0] axorb;
input [size-1:0] a, b;
wire [size-1:0] #delay axorb = a ^ b;
endmodule
```

Пример 40. Задание задержек при назначениях

Минимальная, типовая, максимальная задержка (Minimum, Typical, Maximum Delay)

В языке Verilog HDL выражения задержки могут быть определены как тройное выражение, части которого разделены двоеточиями. Это предпринимается для того, чтобы представить минимальные, типовые и максимальные значения задержки, которые записываются в выражении в том же порядке. Синтаксис такого выражения следующий:

```
<mintypmax_expression>
   ::= <expression>
   ||= <expression1> : <expression2> :
       <expression3>
```

Пример 41. Синтаксис для <mintypmax_expression>

При этом соблюдаются следующие соотношения:

- значение **expression1** меньше или равно значению **expression2**;
- значение **expression2** меньше или равно значению **expression3**.

Синтаксис для задержек на примитивах типа вентилей (включая примитивы, определяемые пользователем), цепи и непрерывные назначения позволяют иметь три значения: каждое для повышающегося фронта, спадающего фронта и задержки на выключение. В примере 42 показано использование задержки со значениям min/typ/max для положительного фронта, отрицательного фронта и задержки на выключение:

```
module iobuf(io1, io2, dir);
                  •.
                  •.
                  •.

bufif0 #(5:7:9, 8:10:12, 15:18:21) (io1, io2, dir);
bufif1 #(6:8:10, 5:7:9, 13:17:19) (io2, io1, dir);
                  •.
                  •.
                  •.
endmodule
```

Пример 42. Пример синтаксиса для задержки

Инструментальные средства для обработки одной модели обычно используют значение по умолчанию для одного набора значений задержки (типовой набор задержек).

Модели в Verilog выполняются, как правило, с тремя значениями для выражений задержки. Три значения позволяют проверить проект с учетом минимальных, типовых или максимальных значений задержки.

В примере 43 выполнена одна из трех указанных задержек прежде, чем моделирование выполняет назначение. **X** примет значение **A** с задержкой, указанной в скобках, и если пользователь не будет выбирать какое-либо одно из них, то симулятор возьмет значение по умолчанию.

```
always @A
  X = #(3:4:5) A;
```

Пример 43. Выполняется одна из трех указанных задержек, прежде чем моделирование выполняет назначение

Значения, выраженные в формате **min:typ:max**, можно использовать и в выражениях. Следующий пример показывает выражение, которое определяет три значения задержки. Минимальное значение — сумма a+d; типовое значение — b+e; максимальное значение — c+f.

```
(a:b:c) + (d:e:f)
```

Часто используемое выражение для задания значения задержки в формате **min:typ:max**:

```
val — (32'd 50: 32'd 75: 32'd 100)
```

Синтаксис для управления задержками в процедурных утверждениях также позволяет выбрать минимальные, типовые и максимальные значения. Они определены выражениями, которые отделены двоеточиями. Пример 44 иллюстрирует это понятие.

```
parameter
min_hi = 97, typ_hi = 100, max_hi = 107;
reg clk;
always
begin
   #(95:100:105) clk = 1;
   #(min_hi:typ_hi:max_hi) clk = 0;
end
```

Пример 44. Управление задержками в процедурных утверждениях

Тип драйвера — Strength

Тип драйвера может быть определен пользователем при непрерывном назначении. Это применяется только к назначениям на скалярные цепи упомянутых далее типов (пример 45).

```
wire wand tri trireg
wor triand tri0
trior tri1
```

Пример 45. Типы драйверов, к которым может быть назначена градация силы драйвера

Непрерывные назначения, определяющие тип драйвера, могут быть сделаны или в объявлении цепи, или автономно в назначении, использующем для этого ключевое слово **assign**. Если цепь имеет признаки градации силы драйвера, то конкретное указание такой градации должно немедленно следовать за ключевым словом (либо используется ключевое слово для типа цепи или ключевое слово при назначении — assign), и градация должна предшествовать любому определению задержки. Всякий раз, когда непрерывное назначение управляет цепью, сила значения моделируется так, как она будет определена.

Спецификация <drive_strength> содержит одно значение силы, которое применяется в том случае, когда значение, назначаемое на

сеть, — 1, и второе значение силы, которое применяется в том случае, когда назначенное значение — 0. Вот как выглядят ключевые слова для определения значения силы для назначения 1:

```
supply1 strong1 pull1 weak1 highz1
```

Пример 46. Ключевые слова для определения значения силы для назначения 1

Следующие ключевые слова применяют для определения значения силы для назначения 0:

```
supply0 strong0 pull0 weak0 highz0
```

Пример 47. Ключевые слова для определения значения силы для назначения 0

Порядок применения этих двух спецификаций типа драйвера произволен. Следующие два правила ограничивают использование спецификаций:

- спецификации типа драйвера (highz1, highz0) и (highz0, highz1) — незаконные языковые конструкции;
- когда ключевое слово **vector** применяется вместе со спецификацией тип драйвера при непрерывном назначении, то ключевое слово **vector** игнорируется.

Продолжение следует