Краткий курс HDL.
Часть 9.
Параметризируемые модели

Иосиф КАРШЕНБОЙМ
iosifk@narod.ru

В этом разделе более подробно рассмотрено создание параметризируемых моделей.

Введение

Для того чтобы многократно использовать модели блоков в различных частях проекта и затем использовать ранее написанные модели блоков в новых проектах, необходимо выполнять модели блоков параметризируемыми. Такие стандартные параметры модуля, как размер — SIZE, разрядность (ширина) — WIDTH и объем (глубина) — DEPTH, обычно выполняются параметризируемыми и могут переопределяться в зависимости от того места в проекте пользователя, где они определены. В описании параметризированных моделей на языке Verilog используют две конструкции — глобальное макроопределение (**`define**) и утверждение для переопределения параметра (**defparam**). В этом разделе будут рассмотрены новые расширения языка Verilog-2001, которые позволяют более удобно выполнять кодирование, а также избежать ошибок при выполнении кодирования.

В Verilog-1995 [см. 2-й пункт списка литературы из части 2], есть два способа определить константы: параметр (**parameter**) — это константа, которая является локальной по отношению к модулю, и макроопределение, созданное путем использования директивы компилятора, — **`define**.

После того, как параметр объявлен, на него можно ссылаться, используя имя параметра.

Так же и на макроопределение **`define** можно ссылаться после того, как оно определено, используя название макрокоманды с предыдущим символом ` (back-tic) апострофа. Различие в проекте между параметром и макроопределением в том, что макроопределение имеет **`identifier_name**, в то время как параметр — только **identifier_name**, без символа апострофа.

Параметры

Параметры должны быть определены в пределах границ модуля, при использовании ключевого слова — **parameter**. Повторим, что параметр — это константа, которая применяется локально в данном модуле и может быть произвольно переопределена при установке данного компонента (instance) в проект пользователя. Для описаний параметризованных модулей, выполненных в стиле Verilog-1995, одно или более объявлений параметров обычно предшествуют объявлениям порта, например, так, как показано в модуле регистра (пример 1).

```
module register (q, d, clk, rst_n);
   parameter SIZE=8;
   output [SIZE-1:0] q;
   input [SIZE-1:0] d;
   input clk, rst_n;
   reg [SIZE-1:0] q;
   always @(posedge clk or negedge rst_n)
   if (!rst_n) q <= 0;
       else q <= d;
endmodule
```

Пример 1. Параметризованная модель регистра, выполненная в стиле Verilog-1995

В варианте исполнения этого же модуля в стиле Verilog-2001 список параметров может быть вынесен в заголовок модуля так же, как это делается в стиле ANSI-C (пример 2).

```
module register2001 #(parameter SIZE=8)
            (output reg [SIZE-1:0] q,
            input [SIZE-1:0] d,
            input clk, rst_n);
   always @(posedge clk, negedge rst_n)
       if (!rst_n) q <= 0;
       else q <= d;
endmodule
```

Пример 2. Параметризированная модель регистра (стиль Verilog-2001)

Параметры и переопределение параметра

Применяя модули с параметрами, выполненными в стандарте Verilog-1995, у пользователя есть два способа изменить параметры как для отдельных модулей, так и для всех примененных в проекте пользователя компонентов для данных модулей.

Первый — это переопределение параметра непосредственно в инстанции компонента, второй — отдельным определением defparam.

В версии Verilog-2001 добавлен третий, более мощный метод для того, чтобы изменить параметры на установленных компонентах модулей: при использовании именованных параметров в непосредственно установленных компонентах модулей (см. часть 7 данной статьи).

Использование символа # при переопределении параметра

Для переопределения параметра в установленном компоненте модуля используется символ #. Он необходим для того, чтобы указать, что параметры данного модуля должны быть переопределены.

В примере 3 две копии регистра из примера 1 установлены в модуле **two_regs1**. Параметр SIZE для обоих случаев устанавливается равным 16, то есть при переопределении параметра назначением # (16) параметр остается тем же самым.

```
module two_regs1 (q, d, clk, rst_n);
   output [15:0] q;
   input [15:0] d;
   input clk, rst_n;
   wire [15:0] dx;
   register #(16) r1(.q(q), .d(dx), .clk(clk), .rst_n(rst_n));
   register #(16) r2(.q(dx), .d(d), .clk(clk), .rst_n(rst_n));
endmodule
```

Пример 3. Переопределение параметра при использовании установленного компонента модуля

Эта форма переопределения параметра поддерживалась всеми инструментальными средствами синтеза много лет. Самая большая проблема с этим типом переопределения параметра состоит в том, что параметры нужно передать в установленный компонент модуля в том же порядке, в котором они появляются в установленном компоненте модуля.

Рассмотрим модуль **myreg** (пример 4).

```
module myreg (q, d, clk, rst_n);
   parameter Trst = 1,
   Tckq = 1,
   SIZE = 4,
   VERSION = «1.1»;
   output   [SIZE-1:0] q;
   input    [SIZE-1:0] d;
   input    clk, rst_n;
   reg      [SIZE-1:0] q;

   always @(posedge clk or negedge rst_n)
       if (!rst_n) q <= #Trst 0;
       else q <= #Tckq d;
endmodule
```

Пример 4. Модуль с четырьмя параметрами

Модуль myreg из примера 4 имеет четыре параметра, и если при установке компонента модуля требуется изменить только третий параметр (например, **SIZE**), модуль не может быть установлен так, чтобы вместо тех параметров, которые не меняются, писались бы только запятые, а для тех параметров, которые мы хотим поменять, вводились бы новые значения. Например, если мы хотим поменять только один параметр **SIZE** так, как показано в примере 5, то это было бы синтаксической ошибкой.

```
module bad_wrapper (q, d, clk, rst_n);
   output [7:0] q;
   input [7:0] d;
   input clk, rst_n;
   // illegal parameter passing example
   myreg #(,,8) r1 (.q(q), .d(d), .clk(clk), .rst_n(rst_n));
endmodule
```

Пример 5. Переопределение параметра таким образом: # (,,8) — это синтаксическая ошибка

Чтобы правильно использовать синтаксис переопределения параметра при установке нового компонента модуля, в нем должны быть перечислены все значения параметров, включая и те значения, которые должны быть изменены. Для модуля **myreg** (пример 4) должны быть перечислены первые два значения параметра, даже при том, что они не изменяются. Новое значение для параметра **SIZE** должно быть задано так, как показано в примере 6.

```
module good_wrapper (q, d, clk, rst_n);
   output [7:0] q;
   input [7:0] d;
   input clk, rst_n;
   // the first two parameters must be
   // explicitly passed even though the
   // values did not change

   myreg #(1,1,8) r1 (.q(q), .d(d), .clk(clk), .rst_n(rst_n));
endmodule
```

Пример 6. Правильный синтаксис при переопределении параметров: # (1,1,8)

Зная об этих ограничениях, инженеры часто перестраивали порядок следования параметров в модуле таким образом, чтобы наиболее часто используемые параметры были помещены в модуле первыми.

Несмотря на ограничения в Verilog-1995 по переопределению параметров, этот метод все же лучше всего поддерживается в компиляторах и симуляторах. И он наболее прост для того, чтобы изменить параметры установленного компонента модуля.

Verilog-2001 фактически расширяет вышеупомянутые возможности по переопределению параметров, добавляя способность передать параметры по имени, подобно тому, как производят подключения портов по их имени.

Скажем «прощай» для defparams!

У многих пользователей языка Verilog первые впечатления, получаемые от применения определений **defparam**, очень благоприятны. Утверждение **defparam** явно идентифицирует установленный компонент модуля и каждый его индивидуальный параметр. При этом каждый индивидуальный параметр модуля должен быть переопределен с каждым утверждением **defparam**. Утверждение **defparam** может быть помещено перед установленным компонентом модуля, после установленного компонента модуля или где-нибудь еще в файле.

До 2000 года инструментальные средства фирмы Synopsys не разрешали использовать утверждения **defparam** для переопределения параметров. И разработчикам, использующим инструментальные средства от Synopsys, необходимо было помнить об этих ограничениях. Но сейчас в последних версиях инструментальных средств Synopsys уже разрешено использовать утверждения **defparam**.

Однако утверждение **defparam** часто может быть неправильно употреблено в следующих случаях:

1. Использование **defparam** для того, чтобы в иерархическом порядке изменить параметры модуля.
2. Размещение определения **defparam** в отдельном файле для изменяемого установленного компонента модуля.
3. Использование многократных определений **defparam** в одном и том же файле, чтобы изменить параметры установленного компонента модуля.
4. Использование многократных утверждений **defparam** в нескольких различных файлах, чтобы изменять параметры установленного компонента модуля.

Иерархические определения defparam

Правильный способ выполнения проектов — это использование определения **defparam** для того, чтобы в иерархическом порядке изменить значения параметров. Это означает, что любой параметр в проекте может быть изменен из любого файла, входящего в данный проект. Потенциально, неправильная запись может вызывать изменение значения параметра модуля, если в установленном компоненте модуля есть определение **defparam** и определение параметра — **parameter**, который, в свою очередь, повторно изменяет параметр в установленном компоненте модуля.

В примере 7 на примере модуля **testbench** (**tb_defparam**) показано то, как изменяется параметр **SIZE** для модуля регистра (в соответствии с параметром **WIDTH**), который передается как параметр **WIDTH** для модуля **dff** (передается к параметру **N**). Модуль **dff** имеет ошибочное иерархическое определение **defparam**, которое изменяет параметр **SIZE** от 8 до 1 в **testbench**, и это значение передается иерархически, чтобы снова изменить в регистре значение **WIDTH** и значения **dffN**.

```
module tb_defparam;
   parameter SIZE=8;
   wire [SIZE-1:0] q;
   reg [SIZE-1:0] d;
   reg clk, rst_n;
   register2 #(SIZE) r1(.q(q), .d(d), .clk(clk),

   .rst_n(rst_n));
   // ...
endmodule

module register2 (q, d, clk, rst_n);
   parameter WIDTH=8;
   output [WIDTH-1:0] q;
   input [WIDTH-1:0] d;
   input     clk, rst_n;

   dff #(WIDTH) d1(.q(q), .d(d), .clk(clk), .rst_n(rst_n));
endmodule


module dff (q, d, clk, rst_n); parameter N=1;
   output [N-1:0] q;

   input [N-1:0] d;
   input clk, rst_n;
   reg [N-1:0] q;

   // dangerous, hierarchical defparam
   defparam tb_defparam.SIZE = 1;

   always @(posedge clk or negedge rst_n)
   if (!rst_n) q <= 0;
   else q <= d;

endmodule
```

Пример 7. Опасное использование иерархического определения defparam

Все порты и переменные в проекте из примера 7 теперь имеют разрядность шины только один бит, в то время как при синтезе модулей **register2** и **dff** их разрядность будет равна восьми битам. Этот тип использования **defparam** вызывает проблемы при проектировании и отладке, и он может легко быть обнаружен.

```
module register3 (q, d, clk, rst_n);
   parameter WIDTH=8;
   output [WIDTH-1:0] q;
   input [WIDTH-1:0] d;
   input clk, rst_n;

   dff3 #(WIDTH) d1
   (.q(q), .d(d), .clk(clk),
   .rst_n(rst_n));
endmodule

module dff3 (q, d, clk, rst_n);
   parameter N=1;
   output [N-1:0] q;
   input [N-1:0] d;
   input clk, rst_n;
   reg [N-1:0] q;

   // dangerous, hierarchical defparam
   defparam register3.WIDTH = 1;

   always @(posedge clk or negedge rst_n)
   if (!rst_n) q <= 0;
   else q <= d;

endmodule
```

Пример 8. Опасное иерархическое использование определение defparam, включенное в модели, — register3/dff3

Пример 8 подобен примеру 7 за исключением того, что определение **defparam** переопределяет разрядность шин в модели **register3** и замыкается само на себя. К сожалению, даже притом, что эта модель при моделировании имеет разрядность в 1 бит, при синтезе она выполняется как модель, имеющая разрядность в 8 бит.

Использование определения defparam в отдельных файлах

Весьма обычны случаи неправильного применения определения **defparams**, когда эти определения помещаются в совершенно другие файлы, а не в тот файл, где находится установленный компонент модуля, который и должен быть изменен.

К сожалению, эта практика была вызвана следующим комментарием в разделе 12.2.1 документов стандарта Verilog-1995 [см. 2-й пункт списка литературы из части 2] и Verilog-2001 [см. 1-й пункт списка литературы из части 2]: «Определения **defparam** особенно полезно сгруппировать вместе, так, чтобы они могли переопределить все параметры в модуле».

Но на самом деле есть превосходная возможность передачи параметров по имени для установленных компонентов модулей, подобно тому, как это делается с определением портов по имени. Таким образом, при использовании передачи параметров по имени, определения **defparam** в конечном счете перестанут использоваться.

Многократное использование определений defparam в одном и том же файле

В предыдущем разделе мы рассмотрели вариант многократного использования **defparam** в разных файлах. Теперь рассмотрим, что произойдет при использовании многократных определений **defparam** в одном и том же файле. Если несколько определений **defparam** помещено в одном и том же файле, и при этом все они изменяют один и тот же параметр, то такое применение определения **defparam** тоже будет неправильно. Стандарт Verilog-2001 определяет правильное действие следующим образом: «В случае многократного использования определения **defparam** для единственного параметра, параметр берет значение последнего определения **defparam**, с которым сталкивается в исходном тексте».

В примере 9 две копии регистра (**register**), приведенного в примере 1, установлены в модуле **two_regs2**. Параметр **SIZE** в обоих случаях установлен равным 16 при использовании определения **defparam**, которое помещено перед соответствующими установленными компонентами регистра (**register**). Третье определение **defparam** располагают после второго установленного компонента регистра, что изменяет разрядность второго регистра. Теперь разрядность второго регистра стала равна 4, а это ошибка.

Поскольку приведенный пример представляет собой очень маленький проект, и компиляторы выдадут сообщение об ошибке (***«Размер порта не соответствует»***), то этот проект нетрудно будет отладить.

К сожалению, часто в больших проектах, где объем RTL-кода измеряется многими

```
module two_regs2 (q, d, clk, rst_n);
parameter SIZE = 16;
output [SIZE-1:0] q;
input [SIZE-1:0] d;
      input clk, rst_n;
      wire    [SIZE-1:0] dx;

defparam r1.SIZE=16;
register r1 (.q(q), .d(dx), .clk(clk), .rst_n(rst_n));

   defparam r2.SIZE=16;
   register r2 (.q(dx), .d(d), .clk(clk), .rst_n(rst_n));
   defparam r2.SIZE=4; // Design error!
endmodule
```

Пример 9. Пример использования определения defparam

страницами, второе «беспризорное» определение **defparam** может быть добавлено по ошибке, и оно переопределит значение параметра. Это произойдет только потому, что проектировщик не обращал внимания на то, что использовалось более раннее определение **defparam**. Такой способ выполнения проектов является более сложным и более трудным в отладке.

Многократное применение определений defparam в отдельных файлах

Иногда определения **defparam** могут быть неправильно многократно размещены в различных файлах.

Большие проблемы вызывает практика размещения многократных определений **defparam** в различных файлах, которые задают значения на один и тот же параметр. Многократные определения **defparam** могут быть обработаны по-разному различными изготовителями программных инструментов, потому что алгоритм обработки для этого сценария никогда не определялся в стандарте Verilog-1995.

Группа разработчиков стандарта Verilog-2001 не хотела поощрять такое поведение, поэтому к стандарту Verilog-2001 была добавлена следующая оговорка: «Когда определения **defparam** находятся во многих исходных файлах, например, они могут быть найдены при поиске библиотечных файлов, то определение **defparam**, по которому параметр берет свое значение, будет неопределенно».

Группа разработчиков стандарта Verilog-2001 хотела пресечь такую практику в целом, поэтому они оставили назначение параметров неопределенными и документально зарегистрировали этот факт, надеясь отговорить любого производителя программных инструментов от поддержки этой некорректной стратегии.

Определения defparam и инструментальные средства

Как уже было показано, определение **defparam** может быть помещено в любом месте проекта, потому что оно может в иерархическом порядке изменить значения параметров любого модуля в проекте. Поэтому применение определений **defparam** создает очень большие трудности при создании программного инструмента, как для продажи пользователям в качестве товарного продукта, так и при применении фирмой для собственных нужд. Эти программные инструменты должны очень точно анализировать проект, в котором разрешено включить определения **defparam**.

Компилятор Verilog не может определить фактические значения любых параметров до тех пор, пока все входные файлы проекта не прочитаны компилятором, потому что чтение последнего файла может изменить каждый отдельный параметр в дизайне.

Поэтому, чтобы уменьшить вероятность появления ошибок в проектах из-за неправильного порядка обработки файлов проекта программными инструментами, можно сформулировать следующее правило: «Не используйте определения **defparam** в любых проектах, выполненных на языке Verilog».

Альтернатива определениям **defparam** приведена в следующем разделе статьи.

Переопределение параметра по имени в стандарте Verilog-2001

К стандарту Verilog-2001 было добавлено следующее расширение: это возможность установки компонентов модулей с параметрами, причем назначение параметра производится по имени параметра, непосредственно в самом установленном компоненте модуля. И теперь это расширение совершенно устраняет потребность в определениях **defparam**.

```
module demuxreg (q, d, ce, clk, rst_n);
   output [15:0] q;
   input      [ 7:0] d;
   input      ce, clk, rst_n;
   wire       [15:0] q;
   wire       [ 7:0] n1;
   not        u0 (ce_n, ce);

   regblk #(.SIZE( 8)) u1
       (.q(n1), .d (d), .ce(ce), .clk(clk), .rst_n(rst_n));
   regblk #(.SIZE(16)) u2
       (.q (q), .d({d,n1}), .ce(ce_n), .clk(clk), .rst_n(rst_n));
endmodule

module regblk (q, d, ce, clk, rst_n);
   parameter SIZE = 4;
   output [SIZE-1:0] q;
   input [SIZE-1:0] d;
   input ce, clk, rst_n;
   reg [SIZE-1:0] q;

   always @(posedge clk or negedge rst_n)
       if (!rst_n) q <= 0;
       else if (ce) q <= d;

endmodule
```

Пример 10. Пример использования передачи параметров по имени

Преимущество новой методики состоит в том, что она предлагает указывать, какой именно параметр будет изменен (подобно тому, как это делает определение **defparam**) и так же, как Verilog-1995, помещает значения переопределенного параметра # в синтаксис установленного компонента модуля.

Поскольку вся информация параметра включена в установленный компонент модуля (инстанцию модели), то такой способ задания параметров абсолютно «понятен» для любого программного инструмента, и это именно та методика, которая должна быть использована при разработке моделей многократного использования.

Правило: *требуйте, чтобы все прохождение параметров в вашем проекте было выполнено в соответствии с новыми методиками переопределения параметра при использовании названий параметров в стиле Verilog-2001.*

Макроподстановка `define (Macro)

Директива компилятора **`define** применяется для того, чтобы выполнить «глобальную» макроподстановку, подобную директиве **#define** в языке Си. Макроподстановки глобальны, начиная от пункта определения, и остаются активными для всего этапа чтения и обработки файлов. Макроподстановки начинаются с того момента, как только это макроопределение сделано, и действуют до тех пор, пока другое макроопределение не изменит значение определенной макрокоманды или же пока макрокоманда не будет отменена использованием директивы компилятора **`undef**.

Макроопределения могут существовать как внутри, так и снаружи объявления любого модуля, и оба эти варианта будут обработаны одинаково. Объявления же параметра могут быть сделаны только в границах модуля.

Начиная с того момента, когда макроопределения будут определены и в течение всего чтения файлов, после того, как макроопределение было определено, использование макроопределений вообще делает компилирование зависимым от порядка расположений файлов дизайна при компиляции.

Типичная проблема, связанная с использованием макроопределений, состоит в том, что несколько файлов могут иметь макроопределения с теми же самыми названиями макрокоманды. Когда это происходит, компиляторы Verilog выдают предупреждения, связанные с «переопределением макрокоманды», но, если разработчик не заметит такое предупреждение, это может привести к большим проблемам при выполнении проекта или отладке.

Почему же плохо переопределить макроопределение? Язык Verilog позволяет иметь иерархические ссылки идентификаторов. Это оказывается очень ценным для исследования и отладки проекта. Если тому же самому названию макрокоманды дать повторное определение в проекте, то только последнее определение будет доступно в **testbench** для исследования и отладки проекта.

Если вы делаете многократные макроопределения на одно и то же название макрокоманды, то вы наверняка полагаете, что макрокоманда должна трактоваться как локальный параметр (**parameter**), а не как глобальная макрокоманда.

Использование `define

Как уже было сказано, компиляторы языка Verilog должны сначала прочитать все макроопределения в проекте и только после этого производить обработку остальных файлов проекта. Соответственно, чтение всех макроопределений в первую очередь при сборке проекта обеспечивает то, что макроопределения будут существовать всегда, когда они необходимы, и что они являются глобально доступными для всех файлов, собранных в проекте.

Обратите внимание на предупреждения о переопределении макрокоманды. Для успешного использования макроопределений можно рекомендовать следующие правила:

- Используйте макроопределения только для идентификаторов, которые однозначно требуют глобального определения идентификатора, который, в свою очередь, не будет изменен в другом месте проекта.
- Где только возможно, поместите все макроопределения в один файл “definitions.vh” и, собирая проект, читайте этот файл первым. ***Примечание:*** *поместите все макроопределения в модуль* ***testbench*** *верхнего уровня и, собирая проект, читайте этот файл первым.*
- Не используйте макроопределения для того, чтобы определить локальные для данного модуля константы.

Включение `define

Перед тем, как использовать макроопределение, обычно выполняется проверка того факта, что само это макроопределение существует. И для этого есть общепринятая методика: в проекте необходимо использовать директиву компилятора **`ifdef** или в новом стиле Verilog-2001 — **`ifndef**. Она делает запрос о существовании макроопределения, сопровождаемого или назначением макрокоманды по **`define**, или **`include** по имени файла, который и содержит требующееся макроопределение (пример 11).

```
`ifdef CYCLE
 // do nothing (better to use `ifndef)
`else
   `define CYCLE 100
`endif
`ifndef CYCLE
   `include «definitions.vh»
`endif
```

Пример 11. Проверка и отладка макроопределений

Директива компилятора `undef

Для того чтобы удалить макроопределение, созданное с помощью директивы компилятора **`define**, Verilog имеет директиву компилятора **`undef**.

Если есть вероятность того, что сделанное в данном файле макроопределение может быть переопределено в другом файле, и, чтобы избежать такой опасной ситуации, нужно сделать следующее. Можно поместить определение **`undef** в конце того файла, где произведено определение макрокоманды по директиве **`define**. Таким образом, использование пары **`define** – **`undef** следует считать самым лучшим средством для решения данной проблемы.

Определение цикла синхрочастоты

Для того чтобы определить периоды синхрочастоты, можно применить либо задание параметров (**parameter**), либо использовать директиву компилятора **`define**. Однако использование **`define** — предпочтительнее.

Правило: *используйте директиву компилятора* ***`define*** *для того, чтобы задать определение тактового сигнала (пример 12).*

Правило: *поместите определения тактового сигнала в файл “definitions.vh” или в* ***testbench*** *верхнего уровня (пример 13).*

```
`define CYCLE 10
```

Пример 12. Определение тактового сигнала

```
`define CYCLE 10
module tb_cycle;
   // ...
   initial begin
   clk = 1'b0;
   forever #(`CYCLE/2) clk = ~clk;
   end
   // ...
Endmodule
```

Пример 13. Глобальное макроопределение тактового сигнала и его рекомендованное описание

Определение длительности цикла синхрочастоты как макроопределения имеет преимущество в том, что синхрочастота — это фундаментальная константа проекта и тестбенча. Цикл общего синхронизирующего сигнала не должен изменяться от одного модуля к другому. Длительность цикла должна быть постоянной!

Профессиональные пользователи Verilog делают большинство симуляций и проверок в тестбенче именно на фронтах синхроимпульса. Поэтому такой тип тестбенча легко масштабируется в соответствии с изменениями в глобальном определении тактового сигнала.

Не применяйте `define в автоматах состояний (FSM)

В некоторых литературных источниках приводятся рекомендации по использованию директивы компилятора **`define** для того, что-

бы определить названия состояний в описании конечного автомата на языке Verilog. Но при разработке конечного автомата использование параметров предпочтительнее.

Для того чтобы определить названия состояний автомата (FSM), целесообразно использовать параметры, именно потому, что названия состояний автомата — это своего рода константы, которые применяются только в модуле, содержащем данный автомат. Если в большом проекте имеется много конечных автоматов, то обычно хочется многократно использовать определенные названия состояний в нескольких описаниях автомата, например RESET, IDLE, READY, READ, WRITE, ERROR и DONE.

Использование директивы **`define** для того, чтобы назначить название состояния, препятствовало бы многократному его использованию. Возможно, в глобальном пространстве имен требуемое вам название уже было взято. Правда, в таком случае можно было бы сделать следующее: применить между модулями к названиям состояния директиву **`undef** и произвести повторное именование состояний по директиве **`define** в новых модулях автомата. Но тогда мы получим множество автоматов с одинаковыми названиями состояний, а это мешает исследовать внутренние назначения шин состояния автоматов в тестбенче и выполнять сравнение диаграмм различных автоматов с одинаковыми названиями состояний. Вот почему нет убедительной причины для определения названий состояний по директиве **`define**. И, следовательно, названия состояний нельзя считать частью глобального пространства имени. Названия состояний нужно счесть местными названиями в том модуле автомата, который их содержит.

Правило: не используйте директиву макроопределения **`define** для назначений названий состояний автомата.

Правило: для назначений символических названий состояний автомата используйте параметры — parameter.

Локальные параметры (localparam) в стиле Verilog-2001

К стандарту Verilog-2001 было добавлено новое расширение, локальный параметр — **localparam**. В отличие от параметров (**parameter**), локальный параметр (**localparam**) не может быть изменен переопределением параметра (выполненное в установленном компоненте как переопределение по позиции или по названию), и при этом **localparam** не может быть переопределен определением **defparam**.

Локальный параметр (**localparam**) может быть определен в терминах параметров, которые могут быть изменены переопределением параметра по позиции в установленном компоненте, переопределением параметра по имени (что является более предпочтительным) или определениями **defparam**.

Идея применения **localparam** в том, что это позволит изменять некоторые местные значения параметров, основанных на других параметрах, защищая **localparam** от случайного или неправильного переопределения конечным пользователем.

В примере14 показано, что размер массива памяти **mem** должен быть сгенерирован в соответствии с разрядностью адресной шины. Разрядность параметра объема памяти **MEM_DEPTH** будет «защищена» от неправильной настройки параметров в том случае, если объявление параметра **MEM_DEPTH** будет помещено в **localparam**. Параметр **MEM_DEPTH** станет иным только при изменении параметра **ASIZE**.

```
module ram1 #(parameter ASIZE=10,

   DSIZE=8)

   (inout [DSIZE-1:0] data,

   input [ASIZE-1:0] addr,

   input en, rw_n);

   // Memory depth equals 2**(ASIZE)
   localparam MEM_DEPTH = 1<<ASIZE;
   reg [DSIZE-1:0] mem [0:MEM_DEPTH-1];

   assign data = (rw_n && en) ? mem[addr]
   : {DSIZE{1'bz}};

   always @(addr, data, rw_n, en)
   if (!rw_n && en) mem[addr] = data;
endmodule
```

Пример 14. Использование локального параметра (localparam) в соответствии со стандартом Verilog-2001. Локальный параметр MEM_DEPTH защищен, он вычисляется в зависимости от значения параметра разрядности адресной шины

Необходимо отметить, что стандарт Verilog-2001 не распространяет возможности расширения **localparam** на список параметров в заголовке модуля. Поэтому в настоящее время **localparam** не может быть добавлен к списку параметров в стиле ANSI, как это показано в примере 15.

```
module multiplier2

   #(parameter AWIDTH=8, BWIDTH=8,

   localparam YWIDTH=AWIDTH+BWIDTH)

   (output [YWIDTH-1:0] y,

   input [AWIDTH-1:0] a,

   input [BWIDTH-1:0] b);

   assign y = a * b;

endmodule
```

Пример 15. Неправомерное использование параметра localparam в заголовке модуля в стиле ANSI

Определение шкалы времени — `timescale

Директива шкалы времени **`timescale** дает значение задержкам, которые могут появиться в Verilog-модели . Шкала времени помещается выше заголовка модуля и имеет следующий вид (пример 16).

```
`timescale time_unit / time_precision
```

Пример 16. Директива шкалы времени

Директива шкалы времени (**`timescale**) имеет огромное влияние на производительность большинства симуляторов языка Verilog. Часто встречающаяся ошибка у новичков при использовании симуляторов состоит в том, что они выбирают **time_precision 1ps** (1 пикосекунда) для того, чтобы тщательно рассмотреть на диаграммах сигналов в своем проекте, изменения за каждую пикосекунду времени моделирования. Применяя при моделировании точность в 1ps, 1ns или 100ps **time_precisions**, время моделирования может увеличиться более чем на 100%, и при этом использование памяти увеличится более чем на 150%.

Довольно часто для того, чтобы облегчить изменение всей шкалы времени в проекте, разработчики используют макроопределение. Все модули в таком проекте включают в себя макрокоманду шкалы времени перед каждым заголовком модуля. В примере 17 показано макроопределение для глобальной шкалы времени (**`timescale**) и использование глобальной макрокоманды шкалы времени (**`timescale**).

```
`define tscale `timescale 1ns/1ns

`tscale

module mymodule (...);

...
```

Пример 17. Глобальное определение макрокоманды шкалы времени (не рекомендуется к применению)

Пользователи обычно надеются управлять эффективностью моделирования, изменяя глобальное определение шкалы времени (**`timescale**), чтобы одновременно иметь возможность изменить и **time_units**, и **time_precisions** для каждой модели, входящей в проект, и, таким образом, повысить производительность симулятора.

Однако необходимо иметь в виду, что глобальное изменение **time_units** каждой шкалы времени (**`timescale**) во всем проекте может неблагоприятно повлиять на его целостность. Любой проект, который включает **#delays**, полагается на точность указанного **time_units** в директиве шкалы времени. В примере 18 требуется, чтобы интервалы времени **time_units** шкалы времени были равны 100ps. Изменение **time_units** до значения в 1ns изменяет задержку от 160ps до 1.6ns, что приводит к ошибке в модели.

```
`timescale 100ps/10ps

module tribuf2001 #(parameter SIZE=8)

   (output [SIZE-1:0] y,

   input [SIZE-1:0] a,

   input en_n);

   assign #1.6 y = en_n ? {SIZE{1'bz}}:a;
endmodule
```

Пример 18. Модуль с time_units по 100ps

Так как величина **time_precision** всегда должна быть равна или меньше, чем **time_unit** в директиве шкалы времени, то при использовании глобальной стратегии по назначению шкалы времени необходимо придерживаться следующих дополнительных рекомендаций.

Правило: *сделайте все определяемые пользователем* ***time_units*** *для шкалы времени равными или большими, чем 1ns.*

Причина здесь в том, что, если в какой-нибудь модели будет использоваться меньшее значение **time_unit**, то, глобально изменяя все значения **time_precision** до величины в 1ns, можно нарушить весь существующий проект.

Необходимо отметить следующее: если модель, поставляемая другими фирмами, будет включена в проект, и если в этой модели использовалось очень маленькое значение **time_precision**, то все моделирование сильно замедлится и его производительность будет крайне низкая, даже в том случае, если в пользовательских моделях значение **time_precision** будет глобально изменено.

Заключение

Макроопределения должны использоваться для того, чтобы определить глобальные константы для всей системы, так, чтобы пользователю было удобно и привычно ими пользоваться, например, названия для команд PCI или глобальных определений тактового цикла.

Каждый раз, как только будет сделано новое макроопределение, название этой макрокоманды уже нельзя безопасно использовать в другом месте в проекте (загрязнение пространства имени). Поскольку в большие проекты для моделирования может быть собрано все больше и больше модулей, то, соответственно, увеличивается и вероятность «столкновения» макроимен. Практика использования макроопределений для констант типа порта или размеров данных и названий состояний может привести к ошибке.

Макроопределения с директивой компилятора **`define** не должны использоваться для определения констант, которые гораздо удобнее могут быть определены в модулях индивидуально.

Параметры **localparam** в Verilog предназначены для того, чтобы представить константы, которые являются для модуля локальными. Локальные параметры имеют дополнительные преимущества в том, что каждый отдельный установленный компонент модуля может иметь различные значения для параметров.

Приведем перечень наиболее важных рекомендаций, приведенных в этом разделе:

- Во всех проектах, выполненных на языке Verilog, не используйте **defparams**.
- Требуйте, чтобы все определение параметров в проекте было выполнено при использовании методики переопределения параметров по имени, в соответствии с новым стандартом Verilog-2001.
- Используйте макроопределения для идентификаторов, которые требуют только глобального определения идентификатора, такого, который не будет изменен в другом месте в проекте.
- Где только возможно, поместите все макроопределения в один файл "definitions.vh" и, собирая проект, читайте этот файл первым. ***Примечание:*** *поместите все макроопределения в модуль* ***testbench*** *верхнего уровня и, собирая проект, читайте этот файл первым.*
- Не используйте макроопределения для того, чтобы определить константы, которые являются местными в конкретном модуле.
- Сделайте определения тактового цикла, используя директиву компилятора **`define**.
- Поместите определения тактовой частоты в файл "definitions.vh" или в **testbench** верхнего уровня.
- Не используйте назначения для названий состояний автомата при макроопределениях **`define**.
- Назначайте состояния, используя параметры parameter с символическими названиями состояний.
- Чтобы улучшать эффективность моделирования, сделайте все определяемые пользователем **time_units** для шкалы времени равными или большими, чем 1ns.

В следующем разделе мы рассмотрим вопросы, связанные с обработкой сигнала «сброс». ■

Литература

1. Cummings C. E. New Verilog-2001 Techniques for Creating Parameterized Models (or Down With `define and Death of a defparam!). Sunburst Design, Inc.