Продолжение. Начало в № 3 `2008

Иосиф КАРШЕНБОЙМ
iosifk@.narod.ru

Краткий курс HDL.
Часть 2. Описание языка

Процедурные назначения (Procedural Assignments)

О непрерывных назначениях уже было сказано. Приведенное здесь описание основных идей подчеркнет различия между назначениями — непрерывными и процедурными.

Итак, непрерывные назначения управляют цепями тем же способом, что и вентили, которые выдают свои выходные сигналы в цепи. И выражения на правой стороне можно трактовать как комбинаторную схему, которая непрерывно управляет цепью. В отличие от непрерывных, процедурные назначения помещают входные значения в регистры, причем эта запись данных производится только в определенные моменты времени. При моделировании само по себе назначение как операция не имеет длительности. Регистр, получив процедурное назначение, держит это значение до следующего процедурного назначения в этот же регистр.

Процедурные назначения происходят в пределах процедур типа **always**, **initial**, **task** и **function** (эти процедуры описаны далее), и их можно трактовать как защелкнутые — «triggered» — назначения. Защелкивание происходит тогда, когда процесс выполнения в моделировании достигает конкретного назначения, находящегося в пределах данной процедуры. Условные утверждения могут управлять процессом назначения: разрешать или не разрешать выполнять блоки назначений. Контроль событий (**events**), управления задержки (**delay**), условные операторы (**if**), операторы выбора (**case**) и утверждения выполнения цикла (**loop**) — всех их можно использовать для того, чтобы иметь возможность управлять процессом выполнения блоков назначений.

Процедурные назначения

Процедурные назначения представляют собой операторы назначения, используемые в функциях. Они подобны непрерывным операторам назначения, описанным в разделе «Непрерывные назначения», за исключением того, что левая сторона процедурного назначения может содержать только переменные типа **reg** и целые числа. Операторы назначения устанавливают значение левой стороны в соответствии с текущим значением правой стороны. Правая сторона назначения может содержать любое произвольное выражение типов данных, описанных в разделе о типах данных, включая простые константы и переменные.

Левая сторона процедурного оператора назначения может содержать только следующие типы данных:

- переменные типа **reg**;
- выбор бита из переменной типа **reg**;
- выборы части битов из переменной типа **reg** (их положение должно быть описано как константы);
- целые числа;
- конкатенации предыдущих типов данных.

Компилятор HDL назначает младший бит на левой стороне в соответствии со значением младшего бита на правой стороне. Если число битов на правой стороне больше, чем число битов на левой стороне, то со старшими битами на правой стороне не производится никаких действий. Если число битов на левой стороне больше, чем число битов на правой стороне, то старшие биты на правой стороне дополняются нулями. Компилятор HDL позволяет выполнять многократные процедурные назначения. В примере 48 показаны некоторые примеры процедурных назначений.

```
sum = a + b; control[5] = (instruction == 8'h2e);
{carry_in, a[7:0]} = 9'h 120;
```

Пример 48. Процедурные назначения

Инициализация переменных (Initial)

В начале симуляции для всех значений переменных задаются значения по умолчанию. При этом вещественные числа инициализируются значением 0, а цепи — значением **z**. Для того чтобы начинать работу симулятора какого-либо конкретного значения, отличного от значения по умолчанию, производится инициализация. Начальное значение выхода для **reg** может быть определено в утверждении инициализации — **initial**.

Инициализация является процедурным назначением и производится только в пределах утверждения **initial**. Это утверждение начинается с ключевого слова **initial**. Утверждение, которое следует за ключевым словом, должно быть оператором назначения, назначающим определенное значение на выход **reg**. В примере 49 показано, как производится инициализация регистра **start**.

```
module shift;
reg [3:0] start;
initial
begin
start = 1;
…
end
endmodule
```

Пример 49. Инициализация регистра **start**

Необходимо также отметить различие между **initial** и **always**. Утверждение **initial** выполняется только один раз при симуляции программы. Утверждение **always** будет выполняться каждый раз, как только изменит свое значение хотя бы одна из переменных, находящаяся в списке чувствительности для этого утверждения.

Блоки always

Разработчик-схемотехник обычно проводит четкое разделение между частью схемы, содержащей только вентили, то есть простую логику, и частью схемы, содержащей триггеры. Однако для языка Verilog не всегда есть возможность задать синтез схем на столь привычном уровне. Компиляторы и симуляторы сами определяют, что именно необходимо синтезировать, в зависимости от того, что разработчик написал в конкретной языковой конструкции. Это замечание необходимо учитывать при работе с блоком **always**.

Блоки **always** могут быть синтезированы как триггеры-защелки или триггеры, срабатывающие по фронту сигнала, но они могут быть выполнены и как комбинационная логика. Блок **always** может содержать логику, выходная информация которой защелкивается на триггерах при изменении уровня сигнала или на переднем/заднем фронте сигнала.

Синтаксис блока **always** показан в примере 50.

```
always @ ( event-expression [or event-expression*] )
   begin
      … statements …
   end
```

Пример 50. Синтаксис блока always

И еще необходимо добавить следующее. Все сигналы, входящие в блоки always, должны быть определены как регистры и триггеры.

Выражение события — event

Выражение события (**event**) объявляет то, что в схеме присутствуют переключающиеся сигналы и осуществляется контроль управления по событиям, вызванным переключением сигналов. Слово **or** объединяет в группы несколько таких сигналов. В языке Verilog определено следующее: если происходят события, указанные в выражениях для переключающихся сигналов, то весь блок выполняется, и это приводит к повторному вычислению всех переменных, входящих в конструкцию **always**. Еще раз необходимо отметить, что

весь блок выполняется даже и в том случае, если только один из переключающихся сигналов в группе имеет событие по переключению — **event**. Однако если выражение для события не происходит синхронно с теми событиями, которые входят в описание конструкции **event**, то компилятор HDL их игнорирует. В примере 51 показан простой пример использования блока **always** с переключающимися сигналами.

```
always @ ( a or b or c ) begin
   f = a & b & c
end
```

Пример 51. Использование блока **always** с переключающимися сигналами

В примере 51 **a**, **b**, и **c** используются как асинхронные переключающиеся сигналы.

Если какой-нибудь из сигналов изменится, то симулятор повторно моделирует блок **always** и вновь вычисляет значение **f**. Однако, на самом деле, в этом примере компилятор HDL не выполнит эти цепи как триггеры, потому что они не синхронны. Но, тем не менее, чтобы получить такую функцию, необходимо указать все переменные, которые используются в блоке **always** как триггеры. Если такое описание не будет сделано, то компилятор HDL выдаст предупреждающее сообщение, подобное следующему.

Предупреждение. *Переменная 'foo' находится в блоке 'bar' (см. строку 88), но она не входит в состав переменных, участвующих в управлении синхронизацией блоком (Warning: Variable 'foo' is being read in block 'bar' declared on line 88 but does not occur in the timing control of the block).*

В редакции Verilog 2001 в качестве выражения для события (**event**) может быть использован символ «*». При этом уже нет необходимости перечислять все сигналы, по изменениям которых происходит повторная обработка конструкции **always**. Код с применением символа «*» приведен в примере 52.

```
module Seg
(output reg eSeg,
input A, B, C, D);
   always @(*)
      begin
            eSeg = 1;
         if(~A & D)
            eSeg = 0;
         if(~A & B & ~C)
            eSeg = 0;
         if(~B & ~C & D)
            eSeg = 0;
      end
endmodule
```

Пример 52. Выражение для события, использующее конструкцию @(*)

Теперь переходим к рассмотрению синхронных триггеров, то есть триггеров, использующих общий сигнал синхрочастоты. В таком случае компилятор HDL «знает», что важны только те состояния триггеров, которые имеются в моменты прихода синхроимпульса. Поэтому для компилятора не нужны перечисления всех переменных в списке чувствительности для блоков **always**, описывающих синхронную обработку данных.

Любой из следующих типов выражений событий (event expressions) может вызвать защелкивание данных в блоке **always**:

- Изменение в указанном значении (пример 53). В этом примере компилятор HDL выполняет цепь не как триггер, а как комбинационную логику.

```
always @ ( identifier ) begin
      … statements …
end
```

Пример 53. Выражение для события — изменение в значении identifier

- По переднему фронту синхрочастоты (пример 54).

```
always @ ( posedge event ) begin
      … statements …
end
```

Пример 54. Выражение для события — передний фронт синхрочастоты

- По заднему фронту синхрочастоты (пример 55).

```
always @ ( negedge event ) begin
      … statements …
end
```

Пример 55. Выражение для события — задний фронт синхрочастоты

- По фронту синхрочастоты или по сигналу асинхронной установки/сброса (пример 56).

```
always @ (posedge CLOCK or negedge reset) begin
   if (!reset) begin
      … statements …
   end
   else begin
      … statements …
   end
end
```

Пример 56. Выражение для события — передний фронт синхрочастоты или по сигналу асинхронной установки/сброса

- По фронту синхрочастоты или по двум сигналам асинхронной установки/сброса, объединенным словом **or** (пример 57).

```
always @ (posedge CLOCK or posedge event1 or negedge event2) begin
   if (event1) begin
      ... statements ...
   end
   else if (!event2) begin
      … statements …
   end
   else begin
      … statements …
   end
end
```

Пример 57. Выражение для события — передний фронт синхрочастоты или по двум сигналам асинхронной установки/сброса, объединенным словом **or**

Когда выражение события не содержит **posedge** или **negedge**, то обычно генерируются не регистры, а комбинационная логика, хотя могут быть сгенерированы проходные триггеры-защелки (flowthrough).

Примечание. *Утверждения* ***@ (posedge clock)*** *и* ***@ (negedge clock)*** *не поддерживаются в функциях (functions) или задачах (tasks).*

Неполное описание события (Incomplete Event Specification)

Есть риск неверного описания блока always, если в нем не перечислены все сигналы, входящие в спецификации события для данного блока. В примере 58 показан неполный список события.

```
always @(a or b) begin
   f = a & b & c;
end
```

Пример 58. Неполный список события

Компилятор HDL строит трехвходовый вентиль AND для описания, приведенного в примере 59. Но при моделировании этого описания переменная **f** не будет повторно вычисляться в те моменты времени, когда будет изменяться сигнал c, потому что этот сигнал не перечислен в выражениях для события. Моделируемое поведение не будет соответствовать поведению трехвходового вентиля AND. В отличие от приведенного примера, моделируемое поведение описания в примере 59 правильно, потому что оно составлено таким образом, чтобы в выражение для события были включены все сигналы.

```
always @(a or b or c) begin
   f = a & b & c;
end
```

Пример 59. Полный список события

В некоторых случаях можно не перечислять все сигналы в спецификации события. И такое описание приведено в примере 60.

```
always @ (posedge c or posedge p)
if (p)
   z = d;
else
   z = a;
```

Пример 60. Неполный список события для асинхронной предварительной загрузки

В логике, синтезируемой в примере 60, сигнал **d** изменяется, как только сигнал **p** будет совершать переход в высокий уровень, и это изменение будет немедленно отражено на выходе **z**. Однако, когда происходит моделирование, сигнал **z** не будет повторно вычислен при изменении сигнала **d**, потому что **d** не перечислен в спецификации события. В результате синтез и моделирование могут не соответствовать друг другу.

Асинхронные предварительные установки и загрузки могут быть правильно смоделированы в компиляторе HDL только тогда, когда вы хотите, чтобы изменения в данных в результате такой загрузки были бы отражены немедленно на выходе. В примере 60 дан-

ные **d** должны измениться на значение, записываемое по предварительной загрузке перед тем, как условие предварительной загрузки (**p**) выполняет переход от низкого уровня к высокому. Если вы пытаетесь читать значение в асинхронной предварительной загрузке, компилятор HDL печатает предупреждение, подобное следующему.

Предупреждение. *Переменная 'd' читается асинхронно в процедуре сброса, строка 21 в файле '/usr/tests/hdl/asyn.v' (Warning:Variable 'd' is being read asynchronously in routine reset line 21 in file '/usr/tests/hdl/asyn.v').*

Таким образом, когда результаты синтеза не совпадают с результатами моделирования, это может вызвать серьезные ошибки в проекте.

Процедурные утверждения назначения — assign и deassign

Процедурные утверждения для назначений **assign** и **deassign** позволяют сделать непрерывные назначения на регистры для тех промежутков времени, которыми управляют эти назначения. Назначение процедурного оператора назначения **assign** отменяет процедурные назначения, заданные на этот регистр до данного момента времени, и производит новые назначения. Процедурное утверждение **deassign** отменяет заданное непрерывное назначение на регистр. Процедурные утверждения **assign** и **deassign** позволяют, например, провести моделирование асинхронных входов сброса/установки (clear/preset) на D-триггере, тактируемом по фронту сигнала. В этом триггере синхрочастота будет запрещена, когда сигналы сброса или установки будут находиться в активном состоянии.

В примере 61 показано использование назначений **assign** и **deassign** в поведенческом описании триггера D-типа с входами сброса/установки.

```
module dff(q,d,clear,preset,clock);
   output q;
   input d,clear,preset,clock;
   reg q;

always @(clear or preset)   // Назначения для
   if(!clear)               // выполнения асинхронного
      assign q=0;           // сброса / установки
else if(!preset)
      assign q=1;           // Назначения для записи
 else                       // входных данных,
      deassign q;           // производящееся по
always @(posedge clock)     // переднему фронту clock
      q=d;                  //- событие @(posedge clock)
endmodule
```

Пример 61. Показано использование назначений **assign** и **deassign** в поведенческом описании триггера D-типа с входами сброса/установки

Если вход сброса или вход установки будет установлен в низкий уровень, то на выходе **q** непрерывно будет выдаваться соответствующее постоянное значение, и положительные фронты синхрочастоты не будут управлять выходом **q**. Когда и вход сброса, и вход установки находятся в высоком уровне, тогда на выход **q** будет действовать назначение **deassign**.

Если ключевое слово назначения **assign** будет применено к регистру, для которого уже есть процедурное непрерывное назначение, то это новое процедурное непрерывное назначение автоматически произведет отмену ранее сделанного на регистр назначения.

Процедурные утверждения force и release

При симуляции проекта иногда возникает необходимость изменить какие-либо сигналы в проекте, но не переделывать сам проект. Например, блок обработки ошибок должен выдавать сигнал по ошибке. Обработка же такой ошибки ведется очень долго по времени симуляции. А мы не хотим «ждать» так долго. Как же можно «пересилить» сигнал ошибки и заставить весь остальной проект обработать данный сигнал? Для этого существует другая форма процедурного непрерывного назначения, и она производится процедурными утверждениями **force** и **release**. Эти утверждения оказывают эффект, подобный действию пары **assign**-**deassign**, но **force** может быть применена к цепям, так же как и к регистрам. Левая сторона назначения может быть регистром, цепью, постоянным выбором бита расширенной векторной цепи, выбором части битов из расширенной векторной цепи или конкатенации. Выражение force не может быть применено к элементу памяти (индексному массиву), быть выбором бита (части битов) из векторного регистра или нерасширенной векторной цепи.

Процедурное утверждение **force**, приложенное к регистру, отменяет процедурное назначение или процедурное непрерывное назначение, которое было сделано на регистр, до тех пор, пока процедурное утверждение **release** не будет сделано на данный регистр. После того, как будет сделано процедурное утверждение **release**, регистр немедленно не изменит свое значение (как это происходит для цепи под управлением утверждения **force**). На выходе регистра будет то значение, которое определено в утверждении **force**, и оно будет сохраняться до тех пор, пока не будет выдано следующее процедурное назначение, кроме того случая, когда процедурное непрерывное назначение активно для данного регистра.

Процедурное утверждение **force**, назначенное на цепь, отменяет действие всех драйверов на этой цепи, таких как выходы вентилей, выходы модулей, и непрерывные назначения — до тех пор, пока на эту цепь не будет назначено процедурное утверждение **release**.

Когда на регистр назначается **release**, и оно в это время имеет активное назначение **assign**, то такое действие вновь установит утверждение **assign**. Как видно из приведенного описания, для регистров существует два уровня назначений и отмены. Причина в том, что **assign**-**deassign** предназначено для описаний аппаратных средств, а **force**-**release** — для того, чтобы проводить отладку.

В примере 62 показан лог от моделирования, в котором в интерактивном режиме выдавались процедурные утверждения **force**-**release**, и как производились «исправления» в выходных сигналах.

```
1   module test;
2     reg
2       a;//=1'hx,x
2       b;//=1'hx,x
2       c;//=1'hx,x
2       d;//=1'hx,x
3     wire
3       e;//=StX
5     and                                  <- AND gate instance
5       and1(e, a, b, c);
7     initial
8       begin
9*        $list;
10        $monitor(`d=%b, e=%b`, d, e)
11        assign d=a&b&c;                  <- assign procedural
12        a=1;                                statement of AND ed
13        b=0;                                values
14        c=1;
15        #10
16          $stop;
16      end
17  endmodule
d=0, e=0
L15 "quasi.v": $stop at simulation time 10
Type ? for help
C1 > faroe d=(a|b|c);                    <- foroe procedural
C2 > faroe e=(a|b|c);                       statements
C3 > #10 $stop;
C4 >
d=1, e=1
C3: $stop at simulation time 20
C4 > release d;                          <- release procedural
C5 > release e;                             statements
C6 > c=0;
C7 > #100 $finish;
C8 > .
d=0, e=0
C7: $finish at simulation time 30
```

Пример 62. Лог от моделирования, в котором в интерактивном режиме выдавались процедурные утверждения force-release

Назначения уровня RTL (RTL Assignments)

В языке Verilog есть одна конструкция, которая для начинающих пользователей представляет довольно серьезное затруднение. Она связана с порядком обработки и назначения сигнала на уровне RTL — передача на уровне регистров. Дело в том, что начинающие пользователи обычно уже имеют некоторый опыт в написании программ на обычных языках программирования, например на ассемблере или Си. Там каждая строка программы обрабатывается последовательно одна за другой. И пользователь привыкает к тому, что написанные им выражения обрабатываются только одно за другим. С другой стороны, разработчики аппаратуры, пришедшие «от схемотехники» к проектированию на Verilog, привыкли к тому, что в электрической схеме прохождение всех сигналов по этой схеме производится только одновременно.

Однако компилятор HDL при обработке переменных может сделать два описанных ранее режима работы, в зависимости от того, как будет сделано описание в проекте. Конечно, если в назначении присутствует только один сигнал, то никакой разницы в обработке данного сигнала не будет. Но когда сигналов два или более, назначения можно выполнить так,

чтобы они не влияли одно на другое, и в таком случае их называют неблокирующими, если же назначения выполняются одно за другим, то они называются блокирующими.

Блокирующие процедурные назначения (Blocking Procedural Assignments)

Процедурный оператор блокирующего назначения должен быть выполнен перед утверждениями, которые следуют за ним в последовательном блоке. Блокирующий процедурный оператор назначения не предотвращает выполнение утверждений, которые следуют за ним в параллельном блоке.

Синтаксис для блокирующего процедурного назначения показан в примере 63.

```
<lvalue> = <timing_control> <expression>
```

Пример 63. Синтаксис для блокирующего процедурного назначения

Здесь выражение **lvalue** имеет тот тип данных, который можно применять для процедурного оператора назначения, знак «=» — это оператор назначения и **timing_control** — это дополнительная задержка внутри назначения. Задержка **timing_control** может быть либо простой задержкой (например, #6), либо задержкой, вызванной управлением по событию (например, **@(posedge clk)**). Значение выражения, находящегося на правой стороне, — **expression** — симулятор назначает на левую сторону.

В примере 64 показаны блокирующие процедурные назначения.

```
rega = 0;
rega[3] = 1;                // a bit-select
rega[3:5] = 7;              // a part-select
mema[address] = 8'hff;      // assignment to a memory element
{carry, acc} = rega + regb; // a concatenation
```

Пример 64. Примеры блокирующих процедурных назначений

Неблокирующее процедурное назначение (The Non-Blocking Procedural Assignment)

Оно позволяет выполнять назначения без блокировки потока выполнения процедурных назначений. Вы можете использовать неблокирующее процедурное назначение всякий раз, когда необходимо выполнить несколько назначений регистра в пределах того же самого шага времени. Причем выполнение каждого из этих назначений не будет зависить от расположения и порядка выполнения для других назначений.

Синтаксис для неблокирующего процедурного назначения показан в примере 65.

```
<lvalue> <= <timing_control> <expression>
```

Пример 65. Синтаксис для неблокирующего процедурного назначения

Здесь выражение **lvalue** имеет тот тип данных, который можно применять для процедурного оператора назначения, знак «<=» — оператор неблокирующего назначения и **timing_control** — это дополнительная задержка внутри назначения. Задержка **timing_control** может быть либо простой задержкой (например, #6), либо задержкой, вызванной управлением по событию (например, **@(posedge clk)**). Значение выражения, находящегося на правой стороне, — **expression** — симулятор назначает на левую сторону.

Оператор назначения неблокирования — тот же самый, что и оператор, применяемый для использования в выражении оператора отношения — «меньше или равно». Программный инструмент сам интерпретирует оператор «<=» в зависимости от того, как этот оператор используется: или для того, чтобы быть оператором отношения, когда он находится в каком-либо выражении, где применяется сравнение, или оператор «<=» интерпретируется как оператор назначения, когда он находится в конструкции неблокирующего процедурного назначения.

Как симулятор обрабатывает и выполняет неблокирующие процедурные назначения

Когда симулятор сталкивается с неблокирующим процедурным назначением, он оценивает и выполняет его за два этапа:

1. Симулятор оценивает правую сторону и намечает назначение нового значения, чтобы выполнить его в то время, которое определено управлением синхронизации для данной процедуры.
2. Симулятор выполняет назначение, поместив значение новых данных на левую сторону либо конце кванта времени, когда задержка, заданная на данную цепь, закончилась, или имело место событие, соответствующее данному оператору. Оба этих шага показаны в примере 66.

Выражение «в конце шага по времени» означает, что неблокирующие назначения, выполняемые во время симуляции «шага по времени», — это последние назначения, выполняемые на данном шаге, но только с одним исключением. События, обрабатывающиеся как неблокирующие назначения, могут создать события для выполнения блокирующих назначений. В этом случае симулятор обрабатывает события блокирующих назначений после выполнения запланированных событий неблокирующих назначений.

В отличие от управления по событиям или задержек, неблокирующие назначения не блокируют процедурный поток. Неблокирующие назначения выполняют назначения, но не останавливают выполнение последующих утверждений в блоке begin-end, как это показано в примере 67.

Примечание. *Как показано в примере 67, симулятор производит назначения в течение конца текущего такта времени и может вы-*

Пример 66. Пример выполнения неблокирующих процедурных назначений

```
module evaluates2(out);

output out;
reg a, b, c;

initial
   begin
   a = 0;
   b = 1;
   c = 0;
   end

always c = #5 ~c;

always @(posedge c)
   begin
   a <= b;  // вычисляет, ждет события
   b <= a;  // и выполняет за два шага
   end
endmodule
```

Шаг 1:
Симулятор вычисляет правую сторону неблокирующего назначения и ждет положительного фронта — **posedge c**, после чего производит назначение переменным новых значений

Шаг 2:
Во время события **posedge c** симулятор обновляет левую сторону каждого неблокирующего назначения во всех утверждениях

Неблокирующие назначения отложены на время 5

a = 0
b = 1

Назначены значения:
a = 1
b = 0

Пример 67. Неблокирующие назначения выполняют назначения, но не останавливают выполнение последующих утверждений в блоке begin-end

```
//non_block1.v
module non_block1(out,);
//input
output out;
reg a, b, c, d, e, f;
//blocking assignments
initial
   begin
   a = #10 1;
   b = #2 0;
   c = #4 1;
   end
//non-blocking assignments
initial
   begin
   d <= #10 1;
   e <= #2 0;
   f <= #4 1;
   end
initial
   begin
   $monitor ($time, ,"a = %b
   b = %b c = %b d = %b e = %b
   f = %b", a,b,c, d,e,f);
   #100 $finish;
   end
endmodule
```

Симулятор назначает значение 1 в регистр **reg a** в момент времени симуляции 10, назначает значение 0 в регистр **reg b** в момент времени симуляции 12 и назначает значение 1 в регистр **reg c** в момент времени симуляции 16

Симулятор назначает значение 1 в регистр **reg d** в момент времени симуляции 10, назначает значение 0 в регистр **reg e** в момент времени симуляции 2 и назначает значение 1 в регистр **reg f** в момент времени симуляции 4

Трассировка неблокирующих назначений

Выполнение отложено до момента времени симуляции 2
e = 0

Выполнение отложено до момента времени симуляции 4
f = 1

Выполнение отложено до момента времени симуляции 10
d = 1

Пример 68. Неблокирующиее процедурные назначения, используемые для операции **swap**

```
//non_block1.v
module
non_block1(out,);
//input
output out;
reg a, b;
initial
   begin
   a = 0;
   b = 1;
   a <= b;
   b <= a;
   end
initial
   begin
   $monitor ($time, ,"a = %b b = %b", a,b);
   #100 $finish;
   end
endmodule
```

Вычисляется, откладывается и выполняется за два шага

Шаг 1
Симулятор вычисляет правую сторону неблокирующего назначения и откладывает выполнение до конца текущего шага по времени симуляции

Шаг 2
В конце текущего временного интервала симулятор обновляет левую сторону каждого утверждения с неблокирующим назначением

Назначены новые значения:
a = 1
b = 0

Пример 69. Когда симулятор выполняет два неблокирующих назначения в один и тот же регистр одновременно, конечное значение регистра не определено

```
module multiple2(out);
output out;
reg a;
initial
   begin
   a <= #4 0;
   a <= #4 1;
   end
endmodule
```

Значение, назначенное в регистры, будет не определено

Заданы для текущего момента времени следующие неблокирующие назначения:
a = 0
a = 1

Значение, назначенное в регистры:
a = ???

полнить операцию обмена данными (swap) при неблокирующих процедурных назначениях.

Когда производятся многократные неблокирующие назначения, которые должны произойти в одном и том же регистре в конкретном временном интервале, то симулятор не может гарантировать порядок, в котором он обрабатывает назначения, при этом конечное состояние регистра будет не определено. Как показано в примере 68, значение регистра a неизвестно.

Если симулятор выполняет два процедурных блока одновременно, и эти процедурные блоки содержат операторы неблокирующего назначения, то конечное значение регистра не определено. Поэтому в примере 69 значение регистра не определено.

Когда многократные неблокирующие назначения с задержками (timing controls) сделаны на один и тот же самый регистр, то эти назначения не отменяют предыдущие неблокирующие назначения. В примере 71 симулятор оценивает оценивает значение **r1** в соответствии со значением **i[0]** и намечает назначения, которые должны произойти после истечения времени для каждой задержки.

Выводы данного раздела: как симулятор производит обработку блокирующих и неблокирующих процедурных назначений.

Каждый раз, в течение интервала времени для моделирования, блокирующие и неблокирующие процедурные назначения будут обработаны следующим образом:

1. Оценивается правая сторона всех операторов назначения в текущем интервале времени.
2. Выполняются все блокирующие процедурные назначения и неблокированные процедурные назначения, которые не имеют никакого управления по синхронизации. В то же самое время неблокирующие процедурные назначения с синхронизацией управления не будут обрабатываться.
3. Симулятор произведет проверку процедур, которые имеют управление по синхронизации, и эти процедуры выполняются, если их выполнение для текущего блока времени разрешено.
4. Симулятор переходит к обработке следующего временного интервала по синхрочастоте.

Пример 70. Обработка двух процедурных назначений одновременно

```
module multiple3(out);
output out;
reg a;
initial a <= #4 0;
initial a <= #4 1;
endmodule
```

Значение, назначенное в регистры, будет не определено

Заданы для текущего момента времени следующие неблокирующие назначения:
a = 0
a = 1

Значение, назначенное в регистры:
a = ???

Блоки утверждений (Block Statements)

Несколько утверждений, находящихся в одном месте описания, можно объединять в группу

В языке Verilog такие группы, следующие друг за другом, должны быть разграничены ключевыми словами **begin** и **end**. В этом языке за ключевыми словами **begin** или **end** не нужно ставить символ **точка с запятой** (;). Для того, чтобы сгруппировать несколько утверждений, пары **begin...end** обычно используются вместе с утверждениями **if**, **case** и **for**. Функции и блоки **always**, которые содержат более чем одно утверждение, требуют, чтобы пары **begin...end** использовались для того, чтобы группировать утверждения.

Последовательный блок имеет следующие характеристики:

- Утверждения выполняются последовательно, одно за другим.
- Задержки являются совокупными; каждое утверждение выполняется после того, как все задержки, предшествующие этому утверждению, завершились.
- Управление симуляцией «выходит» из блока после того, как последнее утверждение будет выполнено.

Блоки, имеющие имя, идентифицируются так: за словом **begin** следует символ двоеточия «:», за ним идет имя блока — block_name, как показано в примере 71. Синтаксис языка Verilog позволяет объявлять переменные локально в названном блоке. Вы можете сделать объявления для **reg**, **integer** и **parameter** только в пределах названного блока, но это невозможно сделать в неназванном блоке. Названные блоки позволяют использовать отключающее утверждение — **disable**.

Формальный синтаксис для последовательного блока показан в примере 72.

Sequential Blocks
<seq_block>
::= begin <statement>* end
|| = begin : <name_of_block>
<block_declaration>*
<statement>*
end
<name_of_block>
::= <IDENTIFIER>
<block_declaration>
::= <parameter_declaration>
|| = <reg_declaration>
|| = <integer_declaration>
|| = <real_declaration>
|| = <time_declaration>
|| = <event_declaration>

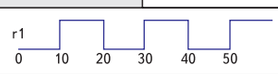
Пример 72. Формальный синтаксис для последовательного блока

В последовательном блоке можно сделать следующие друг за другом назначения, и такие назначения будут иметь детерминированный результат (пример 73).

Пример 71. Симулятор оценивает значение **r1** в соответствии со значением **i[0]** и намечает назначения, которые должны произойти после истечения каждой задержки

```
module multiple;
reg r1;
reg [2:0] i;
   initial
   begin
// starts at time 0
// doesn't hold the block
   for (i = 0; i <= 5; i = i+1)
   r1 <= # (i*10) i[0];
   end
endmodule
```

Сделанные на **r1** назначения не отменяют предыдущие неблокирующие назначения



Отложенное выполнение до времени 0
r1 = 0
Отложенное выполнение до времени 10
r1 = 1
Отложенное выполнение до времени 20
r1 = 0
Отложенное выполнение до времени 30
r1 = 1
Отложенное выполнение до времени 40
r1 = 0
Отложенное выполнение до времени 50
r1 = 1

```
begin
   areg = breg;
   creg = areg; // creg becomes the value of breg
end
```

Пример 73. В последовательном блоке можно сделать следующие друг за другом назначения

В следующем примере сначала будет выполнено первое назначение, и регистр **areg** будет обновлен прежде, чем управление перейдет ко второму назначению.

Задержки можно использовать в последовательном блоке для того, чтобы разнести по времени два назначения (пример 74).

```
begin
   areg = breg;
   #10 creg = areg; // this gives a delay of 10 time
end // units between assignments
```

Пример 74. Задержки можно использовать в последовательном блоке для того, чтобы разнести по времени два назначения

В примере 75 показано, как можно использовать комбинацию последовательного блока и управления задержками для того, чтобы сформировать упорядоченную по времени форму сигнала.

```
parameter d = 50; // d declared as a parameter
reg [7:0] r; // and r declared as an 8-bit register
begin // a waveform controlled by sequential
   // delay
   #d r = 'h35;
   #d r = 'hE2;
   #d r = 'h00;
   #d r = 'hF7;
end
```

Пример 75. Сигнал, которой формируется при помощи задания последовательных задержек

В языке Verilog также применяются конструкции, называемые блоком, имеющим имя, так, как показано в примере 76.

```
begin : block_name
reg local_variable_1;
integer local_variable_2;
parameter local_variable_3;
... statements ...
end
```

Пример 76. Операторы блока, имеющего имя

Блоки, выполняемые параллельно

Блок из нескольких операторов — это средство для того, чтобы собирать в группу два или более утверждений, так, чтобы они действовали синтаксически как единое утверждение. Мы уже ввели и использовали последовательный оператор блока, который ограничен ключевыми словами, **begin – end**.

Второй тип блока ограничен ключевыми словами **fork** и **join**, и этот блок используется для того, чтобы выполнять утверждения параллельно. Блок **fork – join** известен как блок параллельного выполнения и позволяет процедурам выполняться одновременно в течение времени.

Блок параллельного выполнения процедур имеет следующие характеристики:

- утверждения выполняются одновременно;
- значения задержки для каждого утверждения выполняются относительно того времени моделирования, когда управление симулятором начинает работать с данным блоком;
- задержки используются для того, чтобы обеспечить упорядочение времени для назначений;
- управление симулятором «возвращается» из блока, когда выполняется утверждение, последнее по времени моделирования для данного блока, или выполняется утверждение **disable**.

Формальный синтаксис для параллельного блока показан в примере 77.

```
<par_block>
   ::= fork <statement>* join
   ||= fork : <name_of_block
      <block_declaration>*
      <statement>*
   join
<name_of_block>
   ::=<IDENTIFIER>
<block_declaration>
   ::=<parameter_declaration>
   ||=<reg_declaration>
   ||=<integer_declaration>
   ||=<real_declaration>
```

Пример 77. Формальный синтаксис для параллельного блока

В примере 78 показан код, при помощи которого выполняется формирование такого же сигнала, что и в предыдущем примере. Различие в том, что теперь вместо последовательного блока используется параллельный блок.

```
fork
   #50 r = 'h35;
   #100 r = 'hE2;
   #150 r = 'h00;
   #200 r = 'hF7;
join
```

Пример 78. Использование конструкции **fork – join**

Следующие три примера (79–81) используют конструкцию **fork – join**. Все утверждения между ключевыми словами **fork** и **join** выполняются одновременно. Пример 79 показывает состояние гонки сигналов, которая может быть предотвращена при использовании задержки по времени внутри назначения.

```
fork
   #5 a = b;
   #5 b = a;
join
```

Пример 79. Гонка сигналов может быть предотвращена при использовании задержки по времени внутри назначения

В предыдущем примере показан код, когда оба значения — **a** и **b** — должны меняться в одно и то же время моделирования, таким образом создаются гонки фронтов сигналов. Форма назначения, когда задержка находится в правой части, как показано в следующем примере, предотвращает условия для гонки (пример 80).

```
fork // data swap
   a = #5 b;
   b = #5 a;
join
```

Пример 80. Форма назначения, когда задержка находится в правой части и тем самым предотвращает условия для гонки

В этом случае гонка не возникает потому, что задержка, находящаяся с правой стороны, «внутри» назначения, заставляет симулятор сначала оценить значения **a** и **b**, то есть перед тем, как делать задержку, а сами назначения выполнять только после задержки. Такой способ выполнения назначений, но только с ожиданием по событию, также эффективен и не создает гонки. В примере 80 в тот момент, когда симулятор находит операторы назначения, сначала производятся вычисления выражений для правой стороны, а сами назначения будут отсрочены до момента времени, когда поступит передний фронт синхронизирующего сигнала.

```
fork // data shift
   a = @(posedge clk) b;
   b = @(posedge clk) c;
join
```

Пример 81. Симулятор находит операторы назначения

Утверждения if...else

Утверждения **if...else** позволяют выполнять блок утверждений в зависимости от результата проверки на истинность или ложность выражений, определяющих логику выбора.

В соответствии со значением этих выражений и производится обработка блока утверждений, входящих в конструкцию **if...else**.

Синтаксис утверждения **if...else** показан в примере 82.

```
if ( expr )
   begin
   ... statements ...
   end
else
   begin
   ... statements ...
end
```

Пример 82. Синтаксис утверждения **if...else**

Условный оператор состоит из ключевого слова **if** и сопровождается выражением в круглых скобках. Условный оператор **if** сопровождается утверждением или блоком утверждений, которые начинаются со слова **begin** и заканчиваются словом **end**. Если значение выражения является отличным от нуля, то выражение истинно (**true**), и блок утверждений, который следует в этой конструкции, выполняется. Если значение выражения ноль, то выражение ложно (**false**), и блок утверждений, который следует в этой конструкции, не выполняется.

Дополнительное утверждение **else** может следовать за оператором **if**. Если выражение, следующее за **if**, ложно, то выполняется ут-

верждение или блок утверждений, следующий за **else**.

Утверждения **if...else** могут привести к синтезу регистра. Регистры синтезируются в том случае, когда вы не устанавливаете значение этого же регистра **reg** во всех ветвлениях условной конструкции. Компилятор HDL синтезирует логику мультиплексора (или подобную логику выбора) от одного оператора **if**. Условное выражение в операторе **if** синтезируется как сигнал управления на мультиплексор, который определяет выбор сигналов, проходящих через мультиплексор. Например, утверждения в примере 83 создают логику мультиплексора, которая управляет входом c и переключает входы **a** или **b** на выход переменной **x**.

```
if (c)
   x = a;
else
   x = b;
```

Пример 83. Оператор **if**, который синтезируется как мультиплексор

В примере 84 показано, как можно использовать выражение if…else, чтобы создать структуру произвольной длины.

```
if (instruction == ADD)
   begin
   carry_in = 0;
   complement_arg = 0;
   end
else if (instruction == SUB)
   begin
   carry_in = 1;
   complement_arg = 1;
   end
else
   illegal_instruction = 1;
```

Пример 84. Структура выражения **if...else**

В примере 85 показано, как можно использовать вложенные выражения **if...else**.

```
if (select[1])
   begin
   if (select[0]) out = in[3];
      else out = in[2];
   end
else
   begin
   if (select[0]) out = in[1];
      else out = in[0];
 end
```

Пример 85. Структура вложенных выражений **if...else**

Назначения по условию (Conditional Assignments)

В приведенных примерах за выражением **if** всегда следовало выражение **else**. Однако если не делать выражения **else** и не определять состояние сигналов по **else**, то такая переменная будет иметь только одно назначение по условию **if**, и для такой переменной компилятор HDL может синтезировать триггер-защелку. Переменная будет назначена по условию в том случае, если есть хотя бы один путь, который явно не определяет значение на эту переменную. В примере 86 показана переменная **value** управляется по условию. Если **c** не истинно, то переменная **value** не имеет назначения и сохраняет свое предыдущее значение.

```
always begin
   if ( c )
      begin
         value = x;
      end
   Y = value;      // causes a latch to be
                   // synthesized for value
end
```

Пример 86. В данном примере для переменной, управляемой по условию, синтезируется триггер-защелка

Утверждения case

Функции, выполняемые утверждением case, подобны тем, что выполняются в утверждениях **if...else**. Утверждение case позволяет выполнить многократные ветвления цепей в логике на значениях выражения. Один способ описать схему с многократными ветвлениями — с использованием утверждения case (пример 18). Другой способ — с несколькими утверждениями **@ (clock edge)**, которые будут обсуждены в последующих разделах, описывающих применение циклов.

Синтаксис для утверждения case показан в примере 87.

```
case ( expr )
case_item1: begin
... statements ...
end
case_item2: begin
... statements ...
end
default: begin
... statements ...
end
endcase
```

Пример 87. Синтаксис утверждения case

Утверждение **case** состоит из ключевого слова **case**, сопровождаемого выражением в круглых скобках, кроме того, в конструкцию входят одно или более значений для **case** (и утверждения, с ним связанные, которые будут выполняться), сопровождаемые ключевым словом **endcase**. Значения для **case** состоят из выражения (обычно это простые константы) или списка выражений, отделенных запятыми и сопровождаемых двоеточием «:».

При выполнении выражения case производится сравнение значения выражения, следующего после ключевого слова **case**, по очереди с каждым из вариантов значений, находящихся в конструкции **case**. Когда выражение равно значению в одном из вариантов в конструкции, то это условие оценивается как истина (**true**). В каждом из вариантов значений **case** может быть записано несколько выражений, отделенных друг от друга запятыми. Когда используются несколько выражений, то условие будет истинным в том случае, если любое из выражений в теме **case** соответствует выражению после ключевого слова **case**. Первое же значение, которое будет оцениваться как истина в **case**, определит выбор пути. Все последующие значения для **case** игнорируются, даже если они истинны. Если ни одно из значений, находящихся в **case**, не выбрано, то никаких действий не будет выполнено.

Вы можете определить выбор значения в case по умолчанию (**default**) и соответственно приписать ему выражения по умолчанию, которые будут использоваться, когда не выбрано никакое другое значение в **case**. Утверждение **case** показано в примере 88.

```
case (state)
IDLE: begin
   if (start)
      next_state = STEP1;
   else
      next_state = IDLE;
   end
   STEP1: begin
      //do first state processing here
      next_state = STEP2;
   end
   STEP2: begin
      //do second state processing here
      next_state = IDLE;
   end
endcase
```

Пример 88. Утверждение **case**

Дешифратор — Full Case и Parallel Case

В этом разделе будет дано описание языковых конструкций, применяемых для дешифраторов. Такой конструкцией является оператор **case**. Для компилятора языка HDL введены понятия «полный» и «параллельный». Компилятор автоматически определяет, является ли оператор **case** полным или параллельным.

Полный дешифратор. Это значит, что в случае применения оператора **case** будет иметь место полная дешифрация всех состояний, то есть в этом операторе определены все возможные ветвления для выходных сигналов. Если же все возможные ветвления не определены (неполная дешифрация состояний), но все же известно, что одно или более ветвлений никогда не могут происходить, то такой оператор **case** тоже можно объявить полным при помощи директивы **//synopsys_full case**. В противном случае компилятор HDL синтезирует триггер-защелку — **latch**.

Кроме цепей, участвующих в операторе case для передачи сигналов со входа на выход, компилятор HDL синтезирует оптимальную логику для сигналов управления данного узла.

Параллельная дешифрация цепи — это означает, что у сигналов в данных цепях нет перекрытия во времени. Если компилятор HDL принимает решение о том, что никакие другие комбинации сигналов при декодировании невозможны (parallel case), то в таком случае компилятор синтезирует данный фрагмент схемы как мультиплексор. И этот мультиплексор будет представлять собой только комбинационную логику. Можно также объ-

явить оператор **case** как **parallel case** при помощи директивы **// synopsys parallel_case**. В примере 89 показан код, который не приводит к защелке или приоритетному шифратору. Но если компилятор HDL не может распознать ситуацию, что все ветвления параллельны, то он синтезирует такие аппаратные средства, которые будут представлять собой приоритетный шифратор.

```
input [1:0] a;
always @(a or w or x or y or z) begin
   case (a)
      2'b11: b = w ;
      2'b10: b = x ;
      2'b01: b = y ;
      2'b00: b = z ;
   endcase
end
```

Пример 89. Оператор **case**, который является и полным, и параллельным

В следующем примере показан оператор **case**, для которого не определены две комбинации: **2'b01** и **2'b10**. Такой оператор мы будем именовать параллельным, но он не полный. И в таком дешифраторе для сигнала **b** будет установлен триггер-защелка (пример 90).

```
input [1:0] a;
always @(a or w or z) begin
   case (a)
      2'b11: b = w ;
      2'00: b = z ;
   endcase
end
```

Пример 90. Оператор **case**, который является параллельным, но не полным

Оператор case в примере 91 не является параллельным или полным, потому что значения входов w и x не могут быть определены полностью. Однако, если известно, что в определенное время только один из этих входов равняется 2'b11, то есть возможность использовать директиву **// synopsys parallel_case,** чтобы избежать синтеза приоритетного шифратора. Если вы знаете, что или **w**, или **x** всегда равняются 2'b11 (ситуация, известная как неполный дешифратор), то в этом случае для того, чтобы избежать синтеза триггера-защелки, можно использовать директиву **// synopsys full_case**.

```
always @(w or x) begin
   case (2'b11)
      w: b = 10 ;
      x: b = 01 ;
   endcase
end
```

Пример 91. Оператор case, который не является полным и параллельным

Оператор casex

Оператор **casex** позволяет выполнять дешифраторы, аналогично тому, как это делает оператор **case**. Различия между операторами **case** и **casex** не только в ключевом слове, но и в обработке выражений.

Синтаксис для оператора casex показан в примере 92.

```
casex ( expr )
   case_item1: begin
      ... statements ...
   end
   case_item2: begin
      ... statements ...
   end
   default: begin
      ... statements ...
   end
endcase
```

Пример 92. Синтаксис для оператора **casex**

При использовании оператора case можно составлять выражения, состоящие из:

- простых констант;
- списка идентификаторов или выражений, которые отделены запятыми и сопровождены двоеточием «**:**»;
- выражений, состоящих из конкатенации битов или групп битов;
- констант, содержащих биты **z**, **x** или знак «**?**».

Когда символы **z**, **x** или **?** появляются в операторе **case**, это означает, что соответствующий бит выражения **casex** не сравнивался. В примере 93 показано то, как используются символы **x** в операторе **case**.

```
reg [3:0] cond;
casex (cond)
   4'b100x: out = 1;
   default: out = 0;
endcase
```

Пример 93. Оператор casex, в котором применены символы **x**

В примере 93 выход установлен в 1, если **cond** равно 4'b1000 или 4'b1001, потому что последний бит cond определен как **x**. В примере 94 показан сложный фрагмент кода, который может быть значительно упрощен выражением, содержащим **casex** и использующим символ «**?**».

```
if (cond[3]) out = 0;
else if (!cond[3] & cond[2] ) out = 1;
else if (!cond[3] & !cond[2] & cond[1] ) out = 2;
else if (!cond[3] & !cond[2] & !cond[1] & cond[0] ) out = 3;
else if (!cond[3] & !cond[2] & !cond[1] & !cond[0] ) out = 4;
```

Пример 94. Сложный фрагмент кода, не использующий casex с символом **«?»**

В примере 95 показан тот же фрагмент кода, но значительно упрощенный, благодаря использованию casex с символом «**?**».

```
casex (cond)
   4'b1???: out = 0;
   4'b01??: out = 1;
   4'b001?: out = 2;
   4'b0001: out = 3;
   4'b0000: out = 4;
endcase
```

Пример 95. Тот же фрагмент кода, что и в предыдущем примере, но после использования casex с символом **«?»**

Компилятор HDL позволяет использовать символы **?**, **z** и **x** для обозначений битов в выражениях **case**, но не в выражениях casex. В примере 96 показано недопустимое использование выражений в **casex**.

```
express = 3'bxz?;
   ...
   casex (express) //illegal testing of an expression
   ...
endcase
```

Пример 96. Недопустимое использование выражений в **casex**

Оператор casez

Оператор **casez** позволяет выполнять дешифраторы, аналогично тому, как это делает оператор **case**. Различия между операторами **case** и **casez** не только в ключевом слове, но и в обработке выражений. Оператор **casez** работает точно так же, как и **casex**, за исключением того, что символ **x** не разрешен в использовании в этих операторах. Для **casez** разрешены только символы **z** и «**?**».

Синтаксис для оператора casez показан в примере 97.

```
casez ( expr )
   case_item1: begin
      ... statements ...
         end
   case_item2: begin
      ... statements ...
         end
   default: begin
      ... statements ...
         end
endcase
```

Пример 97. Синтаксис утверждения **casez**

При использовании оператора **casez** можно составлять выражения, состоящие из:

- простых констант;
- списка идентификаторов или выражений, отделенных запятыми, сопровождаемыми двоеточием «**:**»;
- выражений, состоящих из конкатенации битов или групп битов;
- констант, содержащих биты **z** или знак «**?**».

Когда обрабатывается оператор **casez**, символ **z** в выражении, обрабатываемом этим оператором, игнорируется. Пример использования оператора **casez** с использованием символов z. показан в примере 98.

```
casez (what_is_it)
   2'bz0: begin
      //accept anything with least significant bit zero
      it_is = even;
      end
   2'bz1: begin
      //accept anything with least significant bit one
      it_is = odd;
      end
endcase
```

Пример 98. Оператор **casez**, в котором применены символы **z**

Компилятор HDL позволяет использовать символы «**?**» и **z** для обозначений битов в выражениях **case**, но не в выражениях **casez**. В примере 99 показано недопустимое использование выражений в **casez**.

```
express = 1'bz;
...
casez (express) //illegal testing of an expression
...
endcase
```

Пример 99. Недопустимое использование выражений в **casez**

Циклы for

Циклы **for** многократно выполняют одно или несколько определений. Повторения выполнены по тому диапазону, который определен границами диапазона и индексом. Два выражения диапазона появляются в каждом цикле **for**: **low_range** — нижняя граница диапазона и **high_range** — верхняя граница диапазона. При этом значение **high_range** больше или равно значению **low_range**. Компилятор HDL распознает как инкремент индекса, так и его декремент. Если надо использовать несколько операторов цикла, то они сопровождаются парой **begin – end**.

Компилятор HDL для цикла for позволяет применять четыре формы синтаксиса, которые показаны в примере 100. В примере 101 показано простейшее применение цикла с оператором **for**. В примере 102 представлено применение вложенных циклов с оператором **for**.

```
for (index = low_range;index < high_range;index = index + step)
for (index = high_range;index > low_range;index = index – step)
for (index = low_range;index <= high_range;index = index + step)
for (index = high_range;index >= low_range;index = index – step)
```

Пример 100. Синтаксис для цикла **for**

```
for (i = 0; i <= 31; i = i + 1)
   begin
      s[i] = a[i] ^ b[i] ^ carry;
      carry = a[i] & b[i] | a[i] & carry | b[i] & carry;
   end
```

Пример 101. Простейший цикл с оператором **for**

```
for (i = 6; i >= 0; i = i – 1)
   for (j = 0; j <= i; j = j + 1)
      if (value[j] > value[j+1])
         begin
            temp = value[j+1];
            value[j+1] = value[j];
            value[j] = temp;
         end
```

Пример 102. Вложенные циклы с оператором **for**

Операторы **for** можно использовать для того, чтобы упростить запись выражений, доступ к которым можно осуществить по индексу. В примерах 103 и 104 показано, как можно оперировать с битами шины.

```
for ( i=0; i < 8; i=i+1 )
   example[i] = a[i] & b[7-i];
```

Пример 103. Пример использования цикла с оператором **for**

```
example[0] = a[0] & b[7];
example[1] = a[1] & b[6];
example[2] = a[2] & b[5];
example[3] = a[3] & b[4];
example[4] = a[4] & b[3];
example[5] = a[5] & b[2];
example[6] = a[6] & b[1];
example[7] = a[7] & b[0];
```

Пример 104. Пример без использования цикла

Циклы while

Циклы выполняют одно или несколько определений до тех пор, пока выражение, определяющее повторения цикла, не примет значение **false**. Циклы **while** выполняют условное ветвление по одному из следующих выражений:

```
@(posedge clock)
```

или

```
@(negedge clock)
```

Компилятор HDL поддерживает выполнение циклов **while** в том случае, если одно из двух выражений будет помещено в выражение для цикла:

```
@(posedge clock)
```

или

```
@(negedge clock)
```

В примере 105 показано неправильное использование цикла с оператором **while**, которое не имеет **event expression**.

```
always
while (x < y)
x = x + z;
```

Пример 105. Неправильное использование цикла с оператором **while**

Если к предыдущему примеру добавить выражение **@(posedge clock)**, то получим выражение, которое будет правильно использовать оператор **while** (пример 106).

```
always
begin @ (posedge clock)
   while (x < y)
      begin
         @ (posedge clock);
            x = x + z;
      end
end
```

Пример 106. Правильное использование цикла с оператором **while**

Циклы forever

Для того чтобы организовать бесконечные циклы, в языке Verilog применяется ключевое слово **forever**. Бесконечный цикл с выражениями **@ (posedge clock)** или **@ (negedge clock)** может быть прерван (to prevent combinational feedback), как показано в примере 107.

Вы можете использовать циклы **forever** с отключающим утверждением, чтобы осуществить синхронные сбросы для триггеров. Отключающее утверждение описано в следующем разделе. Использование стиля, который будет показан в главе 5, в примере 34, — не совсем хорошая идея, потому что нет возможности проверить то, как это работает. Синтезируемый конечный автомат не делает сброс в заранее известное состояние, поэтому невозможно создать тестовую программу для этого случая. В главе 5, в примере 36 будет показано, как можно синтезировать синхронный сброс для конечного автомата.

```
always
forever
   begin
      @ (posedge clock);
      x = x + z;
   end
```

Пример 107. Пример использования цикла с оператором **forever**

Утверждения disable

Компилятор HDL поддерживает утверждение **disable**, которое можно использовать в блоках, имеющих имя **named**. Когда выполняется утверждение **disable**, оно заставляет блок, имеющий имя (named), завершать свою работу. В примере 108 приводится описание компаратора, который использует утверждение **disable**.

```
begin : compare
   for (i = 7; i >= 0; i = i — 1) begin
      if (a[i] != b[i]) begin
         greater_than = a[i];
         less_than = ~a[i];
         equal_to = 0;
         //comparison is done so stop looping
      disable compare;
      end
   end
// If we get here a == b
// If the disable statement is executed, the next three
// lines will not be executed
greater_than = 0;
less_than = 0;
equal_to = 1;
end
```

Пример 108. Описание компаратора, который использует **disable**

В примере 108 описан комбинационный компаратор. Хотя описание кажется последовательным, сгенерированная логика выполняет все действия за один тактовый цикл.

Можно также использовать утверждение **disable**, чтобы осуществить синхронный сброс так, как показано в примере 109.

```
always
forever
begin: Block
   @ (posedge clk)
      if (Reset)
         begin
            z <= 1'b0;
      disable Block;
      end
   z <= a;
end
```

Пример 109. Синхронный сброс регистра при использовании disable в цикле forever

Утверждение **disable**, приведенное в примере 109, заставляет блок (**Block**) немедленно завершить работу и возвратиться к началу.

Утверждения task

В языке Verilog имеются утверждения для выполнения задачи — **task**, они подобны функциям, но утверждения **task** могут иметь порты output и input. Вы можете использовать утверждения **task** для того, чтобы структурировать ваш код в Verilog таким образом, чтобы часть кода могла быть повторно используема при симуляции проекта. Это что-то вроде вызова функций в языках программирования.

В Verilog задачи могут иметь управление синхронизацией и при своем выполнении занимают время, отличное от нуля, перед тем как возвратиться. Однако компилятор HDL игнорирует все управление синхронизацией, таким образом, синтез может не совпасть с результатами моделирования, в том случае, если управление по времени является одной из основных функций схемы.

В примере 110 показано, как утверждение **task** используется для того, чтобы определить функцию сумматора.

```
module task_example (a,b,c);
input [7:0] a,b;
output [7:0] c;
reg [7:0] c;
task adder;
input [7:0] a,b;
output [7:0] adder;
reg c;
integer i;
begin
   c = 0;
   for (i = 0; i <= 7; i = i+1) begin
      adder[i] = a[i] ^ b[i] ^ c;
      c = (a[i] & b[i]) | (a[i] & c) | (b[i] & c);
   end
end
endtask
always
   adder (a,b,c); //c is a reg
endmodule
```

Пример 110. Использование утверждения **task**

Примечание. *Только переменные типа* ***reg*** *могут получить значения выхода от задачи —* ***task****, переменные типа* ***wire*** *этого не могут.*

На этом вторая часть, в которой произведено описание языка Verilog, заканчивается. В следующей части будут приведены примеры кода для наиболее часто встречающихся в проектировании узлов: триггеров, регистров, счетчиков, дешифраторов. ■

Литература

1. IEEE Standard Verilog Hardware Description Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-2001.
2. IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-1995.
3. Bergeron J. Writing Testbenches, Functional Verification of HDL Models. Kluwer Academic Publishers, 2000.
4. Thomas D. E., Moorby P. R.The Verilog® Hardware Description Language. 5th edition. Kluwer Academic Publishers, 2002.
5. Palnitkar S. Verilog HDL: A Guide to Digital Design and Synthesis. 2nd edition. Prentice Hall PTR, 2003.
6. Smith D. HDL Chip Design. 3rd edition, 1997.
7. Madhavan R. Quick Reference for VerilogT HDL. AMBIT Design Systems, Inc.
8. Емец С. Verilog — инструмент разработки цифровых электронных схем // Схемотехника. 2001, № 1–4.
9. Поляков А. К. Языки VHDL и VERILOG в проектировании цифровой аппаратуры. М.: Солон, 2003.
10. Стешенко В. Б. ПЛИС фирмы ALTERA: элементная база, система проектирования и языки описания аппаратуры. М.: Додэка-XXI, 2007.
11. Xilinx Synthesis Technology (XST) User Guide. 1991-2000 Xilinx.
12. Using Verilog to Create CPLD Designs. XAPP143 (v1.0) August 22, 2001.
13. Deepak Kumar Tala. Verilog Tutorial. 2005. **http://www.asic-world.com**