# Verilog – инструмент разработки цифровых электронных схем

### Временной и событийный контроль

Завершая рассмотрение временного и событийного контроля, следует упомянуть о применении intra-assignment delay в "неблокирующем" присвоении, то есть в конструкциях вида

```
x<=#1 y;
a<= @(posedge c) b;
```

Поведение этих конструкций таково, что значение выражения вычисляется и блокирования последовательного исполнения операций не происходит, но новое значение будет присвоено только по истечении времени, указанного во временной конструкции, или после совершения события, указанного в событийной конструкции. В терминах работы программы Verilog симулятора операция присвоения переносится на другой временной шаг. В работе таких конструкций проявляется интересное отличие Verilog симуляторов от VHDL симуляторов. В VHDL каждая следующая по тексту программы операция присвоения одному и тому же сигналу отменяет предыдущую, даже если ее исполнение должно произойти в более ранний момент времени. В Verilog все подобные операции будут помещены в список для соответствующего временного шага, и сигнал, изменение которого вызывают эти операции, будет изменяться в соответствии со всеми операциями. Какой механизм поведения более правильный – вопрос спорный. Так как при синтезе временной контроль игнорируется, да и непонятно, каким образом должна синтезироваться конструкция с присвоением из нескольких источников без специальной разрешающей функции, то это отличие может проявляться только на уровне моделирования с несинтезируемыми элементами. В то же время для работы с несинтезируемыми элементами Verilog предлагает операции, способные отменить (вернее "пересилить") все остальные операции присвоения к одному определенному сигналу. Эти операции присвоения записываются с ключевыми словами force и release.

Проиллюстрировать поведение можно следующим примером:

```
/* это пример на VHDL
LIBRARY ieee;
USE ieee.STD_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY AT91R IS
PORT(NRD: OUT std_logic :='0');
END AT91R;
ARCHITECTURE EBI OF AT91R IS
BEGIN
modeler : PROCESS
B E G I N
NRD <= '1' AFTER 100 ns;
WAIT FOR 30 ns;
NRD <= 'Z' AFTER 30 ns;
WAIT;
END PROCESS modeler;
END EBI;
*/
```

В результате, через 60 нс от начала симуляции, выход переходит в Z-состояние и далее не изменяется. Таким образом, можно видеть, что последующая операция отменяет предыдущую.

Если переписать этот модуль на Verilog без учета изложенного выше, получим (обратите внимание на лаконичность языка Verilog):

```
module AT91R (nrd);
output nrd;
reg ndr;
initial
begin : modeler //пример именованного блока
 ndr<= #100 1'b1;
 #30;
 ndr<= #30 1'bz;
end
endmodule
```

При этом поведение будет другое. Написав соответствую-

щий testbench, можно увидеть:

```
Highest level modules:
tst
0 x
60 z
100 1
```

Это значит, что на 60 нс сработает ndr<= #30 1'bz, а затем на 100 нс расположенный выше по тексту ndr<= #100 1'b1. То есть в Verilog предыдущая операция не отменяется. Для того чтобы поведение модуля было таким же, как и поведение VHDL кода, нужно записать его следующим образом:

```
module AT91R (nrd);
output nrd;
reg ndr;
initial
begin
 ndr<= #100 1'b1;
 #30;
 #30 force ndr = 1'bz;
end
endmodule
```

Использовать force с "неблокирующим" присвоением и/или intra-assignment delay нельзя. Для того чтобы разрешить дальнейшее использование присвоений в других параллельных блоках, сигнал должен быть отпущен с помощью release (например, release ndr;).

Завершая обзор временного контроля, следует упомянуть еще об одной форме задержки – нулевой задержке. В Verilog коде встречаются такие конструкции: #0 a=b. Нулевая задержка означает, что операция будет выполнена в самом конце текущего временного шага. Если в одном временном шаге встречается несколько нулевых задержек, то между собой их порядок не определен.

## Поведенческие конструкции

В поведенческих блоках initial или always могут применяться конструкции управления, сходные с операторами процедурных языков. Данные поведенческие конструкции подразделяются на несколько групп:

1) группа принятия решений: if–else–if , case, casez, casex;
2) группа повторений: repeat, while, for, forever;
3) группа параллельного исполнения: fork-join;
4) оператор wait;

Конструкция if записывается следующим образом:

```
if (<expression>)
 <statement1>
else
 <statement2>
```

Для выбора из нескольких вариантов могут применяться вложенные if.

```
if (<expression>)
 <statement>
else if (<expression>)
 <statement>
else if (<expression>)
 <statement>
else
 <statement>
```

Здесь expression – любое выражение языка, а statement – оператор или группа операторов между begin и end. Ветвь else может отсутствовать, но если имеются вложенные if (как в примере), то else относится к ближайшему if. Для изменения порядка следует пользоваться begin и end. Если получаемое в выражении expression значение не равно 0 и не является неопределенной (x или z), то выполняется ветвь statement1, иначе – statement2. Следует помнить, что так же, как и в языке C, операция сравнения записывается == (два знака "="), в отличие от операции присваивания = (один знак). Но операции сравнения при неопределенных операндах возвращают неопределенное значение (x). Поэтому в поведенческом моделировании (не принимается средствами синтеза) могут использоваться операции === (три знака =) и !==. Эти операции позволяют произвести литеральное сравнение определенных битов в выражении. Еще раз обращаю внимание, что выражение expression не является выражением какого-либо специального типа (boolean), а является любым выражением, которое может быть приведено к типу integer. Здесь прослеживается аналогия с языком C, единственное отличие от которого состоит в том, что Verilog integer может принимать неопределенные значения (x или z). В этом случае выполняется ветвь else.

Исполнение такого кода:

```
module if_test;
initial
begin
 if (2*5) $display("2*5 != 0 ==> true");
 if (2*0) $display("never print this");
  else $display("2*0 != 0 ==> false");
 if (1'bz) $display("never print this");
  else $display("undefined ==> false");
 if (1'bx) $display("never print this");
  else $display("undefined ==> false");
end
endmodule
```

даст следующее:

```
Highest level modules:
if_test
2*5 != 0 ==> true
2*0 != 0 ==> false
undefined ==> false
undefined ==> false
```

Следующий пример иллюстрирует применение операторов сравнения (сколько = в каком случае ставить).

```
module if_test;
reg a,b,c,d;
initial
begin
a=(2'b10>3'b001);b=(2'b10==3'b001);c=(2'b10>2'b0x);d=(2'b10==2'bz0);
$display("a=%b b=%b c=%b d=%b",a,b,c,d);
a=(2'b10!==3'b01);b=(2'b10===2'b10);c=(2'b10!==2'b0x);d=(2'bx1===2'bz1);
$display("a=%b b=%b c=%b d=%b",a,b,c,d);
a=(2'b0x==2'b0x);b=(2'bx1!=2'bx1);c=(2'b0x===2'b0x);d=(2'bx1!==2'bx1);
$display("a=%b b=%b c=%b d=%b",a,b,c,d);
end
endmodule
```

Результат:

```
a=1 b=0 c=x d=x
a=1 b=1 c=1 d=0
a=x b=x c=1 d=0
```

Для выбора из нескольких вариантов также применяется оператор case. Например, данная конструкция реализует дешифратор, подобный К155ИД3.

```
case (rega)
 4'd0: result = 10'b0111111111;
 4'd1: result = 10'b1011111111;
 4'd2: result = 10'b1101111111;
 4'd3: result = 10'b1110111111;
 4'd4: result = 10'b1111011111;
 4'd5: result = 10'b1111101111;
 4'd6: result = 10'b1111110111;
 4'd7: result = 10'b1111111011;
 4'd8: result = 10'b1111111101;
 4'd9: result = 10'b1111111110;
default result = 'bx;
```

```
endcase
```

Оператор case является “непроваливающимся”, в отличие от оператора switch языка C, и гарантирует исполнение одной ветви. В случае если ни одно из условий не совпадает, выполняется ветвь default. Допустимо другое применение – наоборот, в case константа, а в ветвях вычисляемые выражения, либо переменные находятся и там, и там (такого использования оператора выбора в процедурных языках, как правило, нет). Оператор case часто используется в синтезируемом коде для синтеза FSM и мультиплексоров. При этом в несинтезируемых моделях (а в некоторых средствах синтеза и в синтезируемых) в выражениях case могут использоваться литералы с неопределенными значениями. Для поведенческого моделирования используются операторы casez и casex, которые особым образом обрабатывают неопределенные состояния. Синтаксис casez и casex подобен синтаксису case. При этом добавляется символ “?”, используемый в двоичной записи литерала для того, чтобы замаскировать биты, которые не должны влиять на принятие решения.

Для демонстрации обработки неопределенных состояний операторами case, casez и casex рассмотрим следующий пример.

```
module case_test;
integer a,b,c,d;
reg clk;
always #5 clk=~clk;
always
begin : demo
 integer i;
 for(i=0; i<16; i=i+1)
begin
 $write(“i = %d “,i);
 casex (i) // x, z, ? – the same function – ignore bit
  4’b0xxx : $display (“less than 8”);
  4’b10zz : $display (“not less than 8 and less than 12”);
  4’b11?1 : $display (“not less than 12 and odd”);
  4’bx?z? : $display (“other case”);
 default $display (“never print this”);
 endcase
 wait (clk==1’b1); // the same as @(posedge clk)
end
end // demo
initial clk=0;
endmodule
```

Данный пример не содержит $finish, а события будут происходить непрерывно из-за always блоков. Поэтому он будет исполняться “вечно”. Чтобы его остановить, нужно воспользоваться средствами среды.

Фрагмент результата работы:

```
i = 0 less than 8
i = 1 less than 8
i = 2 less than 8
i = 3 less than 8
i = 4 less than 8
i = 5 less than 8
i = 6 less than 8
i = 7 less than 8
i = 8 not less than 8 and less than 12
i = 9 not less than 8 and less than 12
i = 10 not less than 8 and less than 12
i = 11 not less than 8 and less than 12
i = 12 other case
i = 13 not less than 12 and odd
i = 14 other case
i = 15 not less than 12 and odd
i = 0 less than 8
```

Таким образом, case проверяет литеральные совпадения, его можно сравнить с использованием if и === (“=” три раза). В операторах casex и casez биты со значениями x и z или только z игнорируются. В литералах сравнения x и z (casex) или z (casez) могут быть заменены “?”.

Операторы повторения могут встречаться в синтезируемом коде. При этом упрощается и становится более понятной запись. В испытательных стендах и несинтезируемых моделях использование операторов повторения имеет такой же смысл, как и в процедурных языках программирования. Операторы for и repeat были продемонстрированы ранее. Так как Verilog не позволяет воспользоваться вечным циклом языка C (for(;;)), то введен оператор forever. Для выхода из циклов (блоки должны быть именованы) служит оператор disable. Продолжая сравнение с языком C: disable работает, как C операторы break и *continue.*

```
initial
begin :break
for(i = 0;i <n;i =i+1)
begin :continue
 @clk
 if(a == 0) // “continue” loop
 disable continue;
 ... <statements>...
 @clk
 if(a == b) // “break” from loop
 disable break;
...<statements>...
end
end
Еще один цикл while имеет следующую форму:
while (condition)
begin
 statement
 step_assignment;
end
```

Операторы циклов взаимозаменяемы, и выбор определяется личными предпочтениями программиста.

Цикл repeat может использоваться в intra-assignment delay для описания задержки в несколько циклов. Например, таким образом: a = repeat(3)@(posedge clk) b.

Операторы fork–join служат для параллельного исполнения ветвей кода в одном процедурном блоке. Это является несинтезируемой конструкцией и используется редко.

Оператор wait (см. пример casex) используется для приостановки конкурентно исполняемого блока до тех пор, пока не будет выполнено его условие (как правило, элементы выражения условия wait должны изменяться в другом блоке).

## Процедуры и функции

Функции применяются, как правило, для моделирования комбинаторной логики, которую средства синтеза генерируют по описанию функции. В функциях запрещен временной контроль. При вызове функции создается регистр, размерность и имя которого совпадают с размерностью и именем функции. Через этот регистр функция возвращает результат своей работы. Функция может возвращать также целое или вещественное число. Все параметры, передаваемые в функцию, имеют тип input.

```
function [7:0] swap;
input [7:0] byte;
begin
 swap = {byte[3:0],byte[7:4]};
end
endfunction
```

Вызов функции осуществляется следующим образом:

```
a=swap(b);
```

Прежде чем рассматривать процедуры, рассмотрим иерархическую структуру Verilog-модели и области видимости объектов. Локальные переменные (сигналы или параметры)

могут объявляться внутри модуля, именованного блока, процедуры или функции. Если в локальной области видимости симулятору не удается обнаружить переменную, то поиск продолжается в более "высокой" области видимости до тех пор, пока не дойдет дело до переменных, сигналов, параметров, объявленных внутри модуля. Если внутри модуля переменная не обнаружена, то выдается сообщение об ошибке. Таким образом, модуль является высшим элементом иерархии областей видимости. Для доступа к объектам в других модулях, собранных в иерархическую структуру, служит операция разрешения контекста. Эта операция задается с помощью имен модулей или именованных блоков внутри модуля (процедур или функций), разделенных точкой. Так, в примере с делителем частоты (первая часть статьи) для доступа к внутреннему регистру acc модуля NCO_syn из модуля верхнего уровня testbench нужно воспользоваться такой конструкцией <имя включения (instance)>.acc.

```
// 2
always @(negedge clk) $write("Time %t clk %b rst %b f1 %b f2 %b phase1 %b phase2 %b\n", $time, clk, rst, f1, f2, nco1.acc, nco2.acc );
```

При этом на печать будут выдаваться значения фазы (из области видимости модулей NCO_syn)

```
Time 112000 clk 0 rst 0 f1 1 f2 1 phase1 0000 phase2 0000
Time 113000 clk 0 rst 0 f1 1 f2 1 phase1 1011 phase2 1101
Time 114000 clk 0 rst 0 f1 0 f2 0 phase1 0110 phase2 1010
Time 115000 clk 0 rst 0 f1 1 f2 1 phase1 0001 phase2 0111
Time 116000 clk 0 rst 0 f1 1 f2 0 phase1 1100 phase2 0100
Time 117000 clk 0 rst 0 f1 0 f2 1 phase1 0111 phase2 0001
Time 118000 clk 0 rst 0 f1 1 f2 1 phase1 0010 phase2 1110
```

Также можно двигаться вверх или вниз по иерархии включения, используя абсолютные (начинающиеся с модуля высшего уровня), либо относительные имена, что очень похоже на методы работы с файловой системой.

Такой механизм доступа создает ограничение, налагаемое на локальные переменные. Также следует вспомнить о параллельном исполнении и о том, что процедуры в языках HDL не "вызываются", а "разрешаются". Такое название свидетельствует о том, что одновременно может исполняться несколько копий, так как в процедуре разрешен временной и/или событийный контроль. Но при этом память для локальных переменных не выделяется, то есть копии одной и той же процедуры, работающие параллельно, будут "портить" друг другу локальные переменные. Иллюстрацией является практическая невозможность рекурсивных процедур или функций. С этим может столкнуться программист, работавший ранее с процедурными языками.

Синтаксис процедур следующий:

```
task my_task;
input a, b;
inout c;
output d, e;
reg foo1, foo2, foo3;
begin
  <statements> // the set of statements that performs the work of the task
   c = foo1; // the assignments that initialize
  d = foo2; // the results variables
  e = foo3;
end
endtask
```

При этом любое число параметров может передаваться в/из процедуры. Вызов (или "разрешение") процедуры производится следующим образом:

```
my_task (v, w, x, y, z);
```

Функция не имеет права вызывать процедуру, а процедура может разрешать другие процедуры и вызывать себя рекурсивно (см. локальные переменные) или вызывать функции.

## Системные функции

В примерах данной статьи неоднократно использовались системные функции $monitor, $display, $write, $finish, $time. Это малая часть средств, которые предоставляются Verilog системой программисту для анализа результатов моделирования. Благодаря наличию механизма PLI, обеспечивающего подключение исполняемого кода (написанного либо пользователем, либо третьей стороной), число системных функций и задач, которые выполняются с их помощью, очень велико. Основное назначение – это сбор/анализ информации и взаимодействие с системой. Признак системной функции – $. Остановимся на наиболее популярных системных функциях:

$finish – завершение моделирования;

$stop – переход в интерактивный режим;

$display, $write – вывод данных в stdout (дублируется в лог-файл); ведет себя либо как C функция printf с формат строкой (поддерживаются дополнительные форматы, например %b – бинарный), либо как паскалевская процедура write с разделенными запятой аргументами; $display завершает вывод "переводом строки";

$monitor – отслеживает изменения аргументов, в конце каждого временного шага печатает при обнаружении изменения значения; формат как у $display;

$readmemb, $readmemh – обеспечивают считывание данных (в бинарном или шестнадцатиричном представлении) из файла в память (см. первую часть статьи); формат файла очень простой: в каждой строке либо слово требуемой разрядности, либо конструкция @<адрес загрузки>; очень удобно применять для моделирования ПЗУ;

$system – выполняет команду ОС (вызов C функции system());

$fopen, $fclose, $fwrite, $fmonitor – файловые операции, позволяют производить запись в файлы;

$dumpfile, $dumpvars – запись дамп-файлов; позволяют записать изменения сигналов модуля, всего проекта или отдельных в специальном формате для последующего изучения; очень полезный и сильный механизм;

$time – возвращает время симуляции;

$itor, $random – численные функции, выполняют преобразования или возвращают результат математической функции.

Это малая часть стандартных функций. Полный список следует искать в документации к симулятору. Также есть функции, которые не являются стандартными (в настоящий момент времени), но поставляются в виде отдельных объектных модулей или C кода. Примером таких функций является $utConnectivity, записывающая список соединений модели для последующего просмотра с помощью Undertow (http://www.veritools.com/), или $toggle_count, служащая для сбора статистики переключения сигналов.

**Сергей Емец,**
yemets@javad.ru

*Продолжение следует*