(Окончание. Начало — в № 2-4'2001)

Verilog — инструмент разработки цифровых электронных схем

Сергей Емец

yemets@javad.ru

Параметры

При описании цифровых схем возникает необходимость задания констант, определяющих какие-либо фиксированные параметры. Эти константы могут описывать характерные задержки, ширину шин или любой другой параметр, не изменяющийся во время симулирования модели и известный на момент компиляции. Но в то же время при использовании модели одного и того же модуля в различных технологических условиях или включении его различным образом в модули высшего уровня данные константы должны иметь возможность изменяться. В языке VHDL данную функцию решают с помощью конструкции *generic*. В Verilog для этого используются параметры, то есть в описании модуля наряду с описанием переменных и сигналов могут описываться и параметры:

```
module parity (bus, out);
parameter width=8, typ_delay=1, modul_delay=typ_delay*2;
wire [width-1:0] bus;
wire out;
assign #modul_delay out=^bus;
endmodule
```

Данный модуль описывает логику генерации бита четности и может применяться с различной шириной шины и различными задержками. При подключении модуля может сообщаться либо типовая задержка (по которой вычисляется задержка модуля), либо задержка модуля (в случае, когда моделируется нетипичная ситуация).

Включать модуль в иерархию можно следующим образом:

```
parity U1(bus1,out1); // будут использованы значения по умолчанию (width=8, typ_delay=1, modul_delay=typ_delay*2)
parity #(4, 0.5) U2(bus2,out2); // ширина шины — 4 бита, типовая задержка 0.5 (задержка в модуле 0.5*2=1)
parity #(16, 1, 5) U3(bus3,out3); // ширина шины — 16 бит, типовая задержка 1 (игнорируется), задержка в модуле 5
```

Иными словами, конструкция #(фактический параметр, фактический параметр, …) соответствует конструкции *map generic* языка VHDL. Может возникнуть путаница между конструкциями временного контроля и параметрами. Но, во-первых, временной контроль и привязка параметров происходят в разных конструкциях языка и по контексту видно значение знака #, а во-вторых, параметры, как правило, определяют временные задержки внутри модуля. Кроме такого механизма установки параметров существует механизм *defparam* и иерархические имена, позволяющие группировать все фактические параметры в модулях в одном месте вышестоящего модуля. Пример можно переписать следующим образом:

```
module top; //модуль верхнего уровня
…
parity U1(bus1,out1);
parity U2(bus2,out2);
parity U3(bus3,out3);
…
defparam
top.U2.width=4;
top.U2.typ_delay=0.5;
top.U3.width=16;
top.U3.modul_delay=5;
endmodule
```

Так как при описании параметров могут употребляться выражения, данный механизм обеспечивает возможность гибкого повторного использования кода. Следует, однако, помнить, что не всякий синтез поддерживает иерархические имена и *defparam*, поэтому нужно пользоваться первым механизмом для написания синтезируемого кода.

Препроцессор языка Verilog (директивы компилятора)

Наряду с механизмом параметров, описанным выше, в языке Verilog существует механизм, сходный с текстовым *препроцессором* языка C. Этот механизм (по аналогии с C назовем его препроцессором) позволяет проводить предварительную обработку текста до того, как данные будут обработаны симулятором или средством синтеза. Препроцессор обеспечивает условную компиляцию, выполнение макроподстановок, интерфейс с внешней средой. Аналогом условной компиляции Verilog является конструкция if-generate языка VHDL. Для работы данного механизма, а также других директив компилятора используется символ апострофа «`» (не путать с кавычкой в литералах). Наиболее часто применяемые директивы препроцессора — *`define*, *`include*, *`ifdef*, *`else*, *`endif*. Таким же образом задаются директивы компилятора, не являющиеся директивами препроцессора, например *`timescale*.

Для включения текста из одного файла в другой используется *`include* «Имя_Файла» (например, *`include* «dumppins.v»). Следует помнить, что модули Verilog не могут объявляться внутри модуля (не могут быть вложенными), поэтому *`include* можно

использовать либо вне модуля, либо включать в модуль код, не содержащий модулей (*module … endmodule*).

Использование `*define* для описания текстового макроса можно проиллюстрировать следующим образом:

```
`define nc @(negedge clk)
```

Далее в коде программы данный макрос может использоваться следующим образом:

```
always `nc … или a<= `nc b.
```

Пример директив условной компиляции может быть следующим:

```
`ifdef test0
`define CPUTEST0
`include «fault/test0»
`else
`define CPUTEST1
`include «fault/test1»
`endif
```

Поскольку определить макрос можно из командной строки компилятора (опция *+define+_NAME_*), то данный механизм позволяет использовать конфигурационные скрипты. В настоящее время существуют средства синтеза как не поддерживающие препроцессор (старые или недоделанные версии), так и расширяющие стандартный набор директив. Например, средство синтеза может поддерживать директиву `*for*, что позволяет реализовать конструкцию *for-generate* языка VHDL.

Конструкции, применяемые для симулирования библиотечных ячеек

Данная статья направлена на то, чтобы осветить поведенческое моделирование и написание такого поведенческого кода, который может быть синтезирован средством синтеза. Кроме этого, в языке существуют предопределенные модули, которые могут быть использованы как структурные элементы нижнего уровня. Однако некоторые их них практически никогда не используются, поскольку они неприменимы для задач разработки СБИС или ПЛИС. Это, например, описания транзисторных ключей: *nmos*, *pmos*, *rnmos*, *rpmos*, *cmos*, *rcmos*.

Также существуют предопределенные логические ячейки: *and*, *nand*, *nor*, *or*, *xor*, *xnor*, *buf*, *not*, *bufif0*, *bufif1*, *notif1*, *notif0*. Они совпадают по названию с булевыми функциями и являются двухвходовыми элементами, выполняющими соответствующую функцию: *not* — инвертор, *buf* — буфер. Последние четыре элемента имеют вход разрешения и состояние высокого импеданса (*z*). Но так как любой из этих элементов может быть описан с использованием операторов языка, то смысл применения данных конструкций определяется предпочтением разработчика.

Например, *and U1(out,in1,in2)* — то же самое, что *out=in1&in2*; *bufif1 U2 (out, in, control)* — то же самое, что *out=control?in:1'bz*.

Следует заметить, что для облегчения записи элементы могут подключаться на шину или иметь неименованные включения (*instance*), но все же использование операторов кажется предпочтительнее, а результат синтеза или поведение модели будут одинаковыми в обоих случаях.

Для моделирования библиотечных элементов может применяться табличный механизм — UDP (User Defined Primitive). Это несинтезируемая конструкция, похожая по описанию на модуль, но подчиняющаяся более строгим правилам. Назначение UDP — моделировать логику, заданную таблицей истинности. При этом с помощью UDP можно описывать как комбинаторную, так и последовательную логику. UDP может иметь только один выход.

```
primitive and_or(out, a1,a2,a3, b1,b2);
output out;
input a1,a2,a3, b1,b2;
table
//state table information goes here
...
endtable
endprimitive
```

Как видно, основным элементом UDP является таблица истинности. Ее элементы могут принимать следующие значения.

0	Логический 0	
1	Логическая 1	
x	Не определено	
?	Любое значение из 0, 1 и x	Не может использоваться в качестве выходного значения
b	Итерация 0 и 1	Не может использоваться в качестве выходного значения
–	Без изменения	Может использоваться на выходе только последовательного UDP
(vw)	Значение, изменяющееся от v до w	v и w могут быть 0, 1, x,? или b
*	То же, что и ??	Любое изменение значения на входе
r	То же, что и 01	Прямой фронт на входе
f	То же, что и 10	Обратный фронт на входе
p	Итерация (01), (0x) и (x1)	Положительный фронт на входе
n	Итерация (10), (1x) и (x0)	Отрицательный фронт на входе

Количество пробельных символов в описании таблицы не играет роли, важен порядок. Приведем пример комбинаторного UDP (мультиплексора):

```
primitive multiplexer(mux,control,dataA,dataB );
output mux;
input control, dataA, dataB;
table
// control dataA dataB mux
0 1?: 1; //? = 0,1,x
0 0?:0;
1?1:1;
1?0:0;
x 0 0:0;
x 1 1:1;
endtable
endprimitive
```

Пример последовательного UDP:

```
primitive srff (q,s,r);
output q;
input s,r;
reg q;
initial q = 1'b1; // initial statement specifies that output
// terminal q has a value of 1 at the start
// of the simulation
table
// s r q q+
1 0:?: 1;
f 0: 1: -;
0 r:?: 0;
0 f: 0: -;
1 1:?: 0;
endtable
endprimitive
```

Синтезируемое подмножество языка

Прежде чем вести разговор о синтезируемом подмножестве языка, следует остановиться на общих принципах разработки СБИС или ПЛИС с использованием языков высокого уровня. Производитель микросхемы ПЛИС или фабрика, производящая СБИС, предоставляет модель библиотечных элементов, которые могут быть использованы в схеме. Эти библиотеки предназначены для использования в различных областях проектирования (моделирование, синтез, топология кристалла и т. д.) и могут быть выполнены в различных форматах. Один из форматов представляет собой библиотеку элементов, описанных на языке высокого уровня и предназначенных для моделирования. В данной библиотеке содержатся элементы, описанные посредством несинтезируемых конструкций. В описании библиотечных элементов встречаются не рассмотренные в статье конструкции для описания сквозных задержек распространения (*path delay*) и механизмы контроля временных параметров (*setup*, *hold time*). Сквозные задержки позволяют описать задержку модуля, не вдаваясь в его внутреннюю структуру, и поддерживаются специальными словами языка (*specify*, *specparam*, *endspecify*) и специальным синтаксисом (например, (*a,b,c*x>x,y,z*) = *Tin*; (*d+=>x*) = *Tout*;). А системные функции контроля используются для проверки того, происходят ли изменения сигналов в нужные моменты времени, например, для того, чтобы последовательная логика не попадала в метастабильное состояние. Мы не останавливались подробно на этих элементах, поскольку пользователю средств синтеза не нужно описывать библиотечные элементы, а в текстовом описании библиотеки обычно содержится информация о поведении элемента и его временной диаграмме. Можно сказать, что при разработке проекта СБИС или ПЛИС, возможно, даже не придется вручную подключать/отключать эти элементы в структурном описании и вообще иметь сведения об их существовании.

Следующие два элемента разработки предоставляются пользователем — это синтезируемое описание на языке HDL и набор директив для средства синтеза. Также нужен набор средств разработки, включающий в себя симулятор и средство синтеза. Следует остановиться на отличиях синтеза от компиляторов поведенческих языков. Компилятор языка (например C) детерминированным образом пере-

водит операторы языка в команды машинного языка. Синтез переводит последовательные операторы языка HDL в структурную схему, состоящую из библиотечных элементов. Данный процесс больше похож на перебор вариантов и выбор наилучшего, удовлетворяющего временным ограничениям и занимаемой площади. Следствием этого является то, что синтез — итеративный процесс, в котором результаты предыдущего шага используются для коррекции директив для следующего шага. Методология написания директив синтеза и подход к синтезу схемы, имеющей минимальное количество вентилей и максимальную тактовую частоту, является отдельным вопросом и в данной статье не рассматривается. Но в любом случае предоставляемое пользователем описание на языке HDL должно быть правильным — работоспособным и синтезируемым.

Рассмотрим операции, которые необходимо выполнить для разработки прибора на СБИС или ПЛИС:

1) подготовка синтезируемого поведенческого описания схемы (этому шагу может предшествовать моделирование на C, в среде MatLab или с использованием специализированных средств);
2) написание тестовой оболочки — испытательного стенда (*testbench*), — в которой проводится полное тестирование всех режимов модели;
3) итеративная процедура синтеза поведенческой модели, результатом которой является структурная схема (*netlist*) с использованием библиотечных элементов и файл задержек распространения (standard delay file, SDF);
4) проверка работоспособности *netlist*. При этом обычно используется тот же testbench, что и для п. 2. Нарушение работоспособности может быть связано с нарушениями допустимых времен библиотечных элементов, гонками фронтов и пр. В зависимости от опыта разработчика происходит возврат к п. п. 1–3 или переход к п. 5; полезно также на этом шаге проверить результаты работы средств временного анализа (*static timing report*) для выявления «узких мест» — критических путей проекта;
5) процедура размещения и трассировки — выполняется специальными средствами; результатом является коррекция SDF-файла (помимо прошивки ПЛИС или разводки СБИС);
6) проверка *netlist* с новыми задержками; подобна п. 4, но возврат возможен к п. п. 1, 3, 5;
7) *in-place* оптимизация (для большинства современных ПЛИС отсутствует), в ходе которой производится выравнивание времен распространения сигналов — усиление/ослабление выходов ячеек, установка буферов/элементов задержки;
8) повтор п. 6 до тех пор, пока не будут достигнуты требуемые характеристики;
9) подготовка производственных тестов, предназначенных для поиска неисправностей в СБИС (для ПЛИС этот шаг необязателен).

Шаг 3 в данной схеме раньше выполнялся вручную, когда поведенческое описание заменялось на структурное самим разработчиком. В настоящее время существуют средства синтеза, которые позволяют получить более эффективную схему. Пользование этими средствами накладывает ограничение на употребление конструкций языка в исходном коде. Требуется синтезируемая модель. Кроме того, чтобы многократно не переписывать поведенческую модель, следует четко представлять, какое Verilog-описание приведет к синтезу того или иного элемента схемы.

Ранее в статье было продемонстрировано, как работает Verilog-симулятор. Далее будут рассмотрены требования средств синтеза.

В результате синтеза любой элемент языка может быть синтезирован, проигнорирован, либо может вызвать ошибку. Конструкции языка могут поддерживаться полностью, частично или не поддерживаться вовсе.

Как уже упоминалось, иерархические имена, *initial*, *fork-join* или *primitive*, не поддерживаются. Временной контроль *#nn* игнорируется в синтезируемой модели. Событийный контроль поддерживается частично: только в блоках *always*.

В документации к средству синтеза указывается, какие ограничения вводятся на элементы языка. Перед написанием модели следует ознакомиться с документацией.

Кроме этого, существуют правила описания комбинаторной логики, последовательной логики, мультиплексоров, FSM и т. п.

Комбинаторная логика синтезируется из следующих конструкций:

1) непрерывное присвоение:

```
assign a=b+c&d;
wire b={e,f} | g[1:0];
```

2) сигналы, описанные в функциях;
3) сигналы, описанные следующей или подобной конструкцией:

```
reg data_out;
always @(a or b or c)
if (b)
data_out = a;
else
data_out = c;
```

то есть блок *always*, в списке чувствительности которого перечислены все входные сигналы.

Как можно видеть в третьем (а иногда и во втором) случае, объявление сигнала с ключевым словом *reg* не приводит к созданию регистра. Также комбинаторная логика синтезируется в случае *case*, если описаны все ветви. При этом получается не приоритетный набор (как в моделировании), а набор параллельных конструкций. Также *case* используется для синтеза мультиплексоров.

Последовательная логика имеет ограничения в синтезируемых конструкциях. Для описания регистра-защелки (latch) применяется следующая конструкция:

```
reg data_out;
always @(data_in or enable)
if (enable)
data_out = data_in;
```

Для регистров, работающих по фронту/срезу сигнала, применяется описание с конструкциями *posedge* или *negedge*:

```
reg data_out;
always @(posedge clock)
data_out = data_in;
```

Чтобы добавить синхронный сброс, описание нужно дополнить сигналом установки/сброса, но не вносить эти сигналы в список чувствительности:

```
reg data_out;
always @(posedge clock)
if (set_sig)
data_out = 1'b1;
else if (reset_sig)
data_out = 1'b0;
else
data_out = data_in;
```

Для асинхронной установки/сброса конструкция должна быть изменена, так чтобы эти сигналы попали в список чувствительности:

```
reg data_out;
always @(posedge clock or posedge set_sig or posedge reset_sig)
if (set_sig)
data_out = 1'b1;
else if (reset_sig)
data_out = 1'b0;
else
data_out = data_in;
```

Пользуясь этими принципами, можно описать триггер любого типа.

Если сигнал имеет размерность, большую единицы, то синтезируются устройства для каждого бита, а существующая логика или арифметика в подобных конструкциях синтезируется в комбинаторную схему.

Блок *case*, у которого расшифровываются не все входные воздействия, синтезируется в элемент последовательной логики. Также может использоваться для описания FSM (finite state machine). Поэтому кажется более правильным воспользоваться комбинаторным *case* и явно описать регистр для хранения результата (к примеру, для реализации FSM).

Ограничения, накладываемые средством синтеза на язык HDL, позволяют описывать логику работы схемы в виде комбинаторной логики и регистров для хранения результатов. Такое описание называется RTL (register transfer level) и иногда отделяется (по смыслу) от поведенческого. Основная идея, которую хотелось бы высказать в данной статье, — это то, что следует пользоваться RTL-описанием для моделирования и синтеза. Данный подход является эффективным методом ведения разработки. Так как «правильное» (написанное с применением правил, изложенных выше) поведенческое описание может быть автоматически синтезировано на уровень вентилей, это позволяет значительно ускорить и упростить процесс разработки. При этом, если временные ограничения были выбраны правильно (с учетом задержек в элементах библиотеки), поведение RTL (поведенческой модели) не будет отличаться от поведения нетлиста (gate-level) и не должно отличаться от поведения готового изделия.

Завершая рассмотрение синтезируемого (RTL) кода, следует упомянуть о директивах синтеза, не являющихся элементами языка Verilog. Они задаются в поле комментария и игнорируются при моделировании, но управляют средством синтеза, например:

```
// synopsis synthesis off
$write(«this string is ignored by synthesis»);
// synopsis synthesis on
```

Для уже упоминавшегося *case* существует директива синтеза (выбирающая синтез в последовательную или параллельную логику), которая может быть указана в поле комментария:

```
// ambit synthesis case = full | parallel | mux
```

Однако данный механизм нестандартен и его использование оправдано в редких случаях.

Заключение

Данная статья не является документацией языка Verilog. Целью ее написания было проиллюстрировать работу Verilog-симулятора и показать возможность использования языка для разработки цифровых схем. При этом рассматривались как аспекты, связанные с тестированием модели (написание испытательных стендов), так и проблемы написания синтезируемого кода. Приведенные примеры и объяснения к ним должны были (по замыслу автора) показать различия между процедурными языками и языками описания HDL. Развитие CSoC и ПЛИС позволяет предположить, что все большие части проекта могут быть реализованы в виде описания HDL. В статье была сделана попытка показать, что Verilog является сильным и удобным средством для достижения этой цели.