Verilog — инструмент разработки цифровых электронных схем

Сергей Емец

yemets@javad.ru

Введение

Язык Verilog был разработан фирмой Gateway Design Automaton как внутренний язык симуляции. Cadence приобрела Gateway в 1989 г. и открыла Verilog для общественного использования. В 1995 г. был определен стандарт языка — Verilog LRM (Language Reference Manual), IEEE1364-1995. Таким образом, датой появления языка Verilog следует считать 1995 г. К этому времени уже успел получить широкое распространение другой язык высокого уровня для описания принципиальных схем — VHDL (Very high-speed IC Hardware Description Language), появившийся еще в 1987 г. Несмотря на похожие названия, Verilog HDL и VHDL — различные языки. В статье сравниваются конструкции этих языков, предполагая, что читателю известен VHDL, но данный материал будет также полезен и для начинающих изучение HDL с языка Verilog. По мнению автора, Verilog — достаточно простой язык, сходный с языком программирования C как по синтаксису, так и по «идеологии». Малое количество служебных слов и простота основных конструкций упрощают изучение и позволяют использовать Verilog в целях обучения. Но в то же время это эффективный и специализированный язык. VHDL обладает большей универсальностью и может быть использован не только для описания моделей цифровых электронных схем, но и для других моделей (например, модели экосистемы). Однако из-за своих расширенных возможностей VHDL проигрывает в эффективности и простоте, то есть на описание одной и той же конструкции в Verilog потребуется в 3–4 раза меньше символов (ASCII), чем в VHDL. В Verilog существуют специфические объекты (UDP, Specify-блоки), не имеющие аналогов в VHDL. Также следует упомянуть стандарт PLI (Program Language Interface), который позволяет включать функции, написанные пользователем (например, на C), в код симулятора. В настоящее время важным недостатком Verilog является отсутствие документации, особенно на русском языке. Данная статья является попыткой улучшить эту ситуацию.

Основы синтаксиса

Для иллюстрации примеров будет использоваться Verilog-XL симулятор от Cadence (**ww w.cadence.com**). Это наиболее стандартный, если так можно выразиться, симулятор. Существует несколько проектов бесплатных симуляторов Verilog. Автору известны Veribest (**ht tp://www .flex.c om/~jching**) и Icarus Verilog (**ht tp://www .icarus.com/eda/verilog /index.html**). Однако в настоящее время данные продукты не являются завершенными, и их работоспособность на больших проектах, полнота и правильность поддержки стандарта являются неудовлетворительными. Но для примеров, рассмотренных в статье, эти программы вполне подходят. Еще одним неприятным моментом является то, что собрать эти симуляторы для работы под Windows автору не удалось (пакеты поставляются в исходном коде и в скомпилированном виде для ОС Solaris и Linux). Большинство производителей также предлагают бесплатные демонстрационные версии симуляторов Verilog. Неплохой список ссылок на страницы с подобным программным обеспечением находится по адресу **ht tp:// sal. kachinatech. com/Z/1/index.shtml**. Для Windows-платформы весьма популярен ModelSim (**ww w.model .com**). Под Linux хорошо зарекомендовал себя finsim (**ht tp://w ww.fintronic .com**), для которого существует и Windows-версия. Также в большинстве современных продуктов, предназначенных для синтеза и верификации проектов на FPGA или ASIC, существует поддержка Verilog HDL и VHDL как для синтеза, так и для симуляции.

Симуляторы можно условно разбить на две группы: интерпретирующие и компилирующие в «родной код». Так, Verilog-XL является интерпретирующим симулятором, NC-Verilog — компилирующим, а finsim может работать как в том, так и в другом режиме. Для работы компилирующих симуляторов обычно требуется, чтобы в системе был установлен C-компилятор.

Запуск симулятора может производиться из командной строки или из графической оболочки, которая формирует строку для запуска. Часть параме-

тров, передаваемая в командной строке, уникальна для конкретного симулятора, но есть основной стандартный набор. Так, вызов симулятора со списком файлов вызовет их исполнение (Verilog Test.V Modul.V). Список файлов можно поместить в отдельный файл и вызывать симулятор с опцией -f (Verilog -f FileList).

Все приводимые в тексте статьи примеры можно загрузить по адресу **http://chat.ru/~verilog_intro**.

Внимание: данное описание не является полным изложением синтаксиса языка; в статье рассмотрены только основные конструкции, необходимые для понимания структуры и принципов языка.

Типы данных

Verilog поддерживает следующие «стандартные» типы данных: целое — integer (32-битовое со знаком) и real — число с плавающей точкой.

Для моделирования также используются time (время), специфический тип, применяемый встроенными функциями для моделирования времени, обычно 64-битовое целое; event (событие) — в языке существует ряд операторов и конструкций для работы с событиями.

Создавать свои типы данных, как в VHDL, нельзя. При разработке синтезируемых моделей из перечисленных типов используется только integer.

Сигналы, в отличие от VHDL, бывают двух основных типов: «цепи» и «регистры». Самые распространенные из них описываются ключевыми словами wire и reg соответственно. Однако следует помнить, что средство синтеза не всегда реализует reg в виде триггера. Отличие wire от reg состоит в том, что reg способен сохранять присвоенное значение (работает как переменная в языках программирования), а к wire требуется прилагать непрерывное воздействие (driver). То есть wire моделирует провод, который переходит в неопределенное состояние при отключении драйвера. Существуют также wand, wor, tri0, tri1, triand, trior и trireg (это цепь, а не регистр!) для моделирования различных типов цепей (wand — wired and, tri0 — резистор к 0, trireg — емкость и т. п.), но такие цепи встречаются редко и потому в статье не рассматриваются.

Идентификаторы в Verilog являются чувствительными к регистру написания и подчиняются обычным правилам: не могут начинаться с цифры или знака $ и могут содержать буквы, цифры, «$», «_». Существуют так называемые escaped-идентификаторы (в основном в структурных моделях, полученных после синтеза), которые начинаются с «\», содержат любые символы и заканчиваются пробелом или переводом строки.

Пример:

```
// — это комментарий
/* — и это комментарий
*/
integer i, j, k;   // объявление переменных i, j и k типа integer
time start, duration;
real freq_div;
event start_process;
/* далее следуют объявления
однобитовых сигналов */
wire a, b, c;
wire d;
reg store, ff, A;  // reg A не совпадает с wire a (case sensitive vs VHDL)
reg \dut/cntr/reg_s ; // escaped идентификатор (пробел перед «;»)
```

Для описания шин или регистров неединичной ширины используются диапазоны (range) вида [n:m], где m и n — целые числа или параметры. В языке допускается как m>n, так и наоборот. Но это имеет значение в операциях, для которых важен порядок битов (например, сложение или присвоение целого). Поэтому принято располагать индексы в убывающем порядке:

```
wire [7:0] data_bus;
reg [3:0] high_nibble, low_nibble; // два 4-битовых регистра
reg [0:5] a_reg; // регистр с обратным порядком битов не рекомендуется
```

Массивы в Verilog не поддерживаются, но существуют «памяти», собственно одномерный массив или модель памяти:

```
reg [8:0] Fifo [31:0]; // 32 слова 9-битовой памяти
```

В отличие от VHDL в Verilog представление сигналов реализовано в самом языке (а не в библиотеке). Всего существуют четыре типа значений, которые могут принимать сигналы «цепь» и «регистр»: 0, 1, z, x. Первые три соответствуют логическим уровням и состоянию с высоким импедансом. Четвертый (x) означает неопределенное состояние и используется при моделировании неинициализированных сигналов, конфликтов (два выхода с противоположными состояниями соединены вместе), метастабильных состояний триггеров (при нарушении временных соотношений между входами данных и тактовым входом), иными словами, во всех случаях, когда симулятор не может определить значение данного сигнала. В реальном приборе такого сигнала, конечно, не бывает.

Для записи многоразрядных сигналов (констант) используются следующие конструкции: 1'bz — одноразрядный высокоимпедансный сигнал, 10'd1_000 — десятиразрядное число 1000, записанное в десятичной системе (символ «_» игнорируется), 4'bx01z — четырехразрядный сигнал с неопределенным старшим битом, высокоимпедансным младшим и вторым и третьим, равными 1 и 0 соответственно. То есть запись многоразрядного сигнала представляет собой разрядность, одинарную кавычку «'» (не путать с апострофом «`», использующимся в директивах), основание системы счисления (b, o, d, h) и цифры, использующиеся в данной системе счисления. В двоичной системе допустимо использование символов z и x. Символ подчеркивания служит для облегчения записи и игнорируется. Использование констант без указания разрядности не желательно, так как по умолчанию константа воспринимается с длиной 32 бита. Данные типа integer также могут присваиваться регистрам.

Структурное описание

Основной структурной единицей Verilog описания является module. Модуль соответствует entity в VHDL. Модуль описывается ключевыми словами module — endmodule. В файле может быть описано несколько модулей. Другие модули могут подключаться к цепям модуля, образуя иерархическую структуру. При запуске Verilog симулятор строит иерархическое дерево из всех модулей, которые обнаружены в файлах, поданных на вход симулятора, и находит модуль верхнего уровня. Если таких модулей несколько, то происходит ошибка. Как правило, модуль содержит список портов — интерфейсных сигналов, которые служат для подключения его в других модулях. Порты бывают трех типов input — входы, output — выходы, inout — двунаправленные. Входы и двунаправленные порты должны иметь тип wire, а выходы могут быть как wire, так и reg.

Синтаксис модуля рассмотрим на примере накопительного сумматора:

```
module NCO (FC, CO, C); // имя модуля и список портов
input FC, C;                    // входы
output CO;// выход
// описание используемых сигналов
wire [3:0] FC;
wire C; // не обязательно, так как по умолчанию вход является однобитовым проводом
reg [3:0] acc;
reg CO;
// описание поведения системы
initial
 begin
 acc=0;
 CO=0;
 end
always @(posedge C)        // событие — фронт C
                {CO,acc}={CO, acc}+FO;
endmodule
```

Модуль NCO не включает в себя другие модули и является модулем нижнего уровня иерархии. В модуле присутствуют две «поведенческие» конструкции: initial и always. Initial служит для описания действий, которые выполняются один раз (при запуске модели), а always обозначает действия, которые выполняются постоянно. Ключевые слова begin/end имеют такое же значение, как в процедурном языке Паскаль (соответствуют {} в C). Для того чтобы always имело смысл, используется событийный контроль — конструкция @(posedge C), означает «по положительному фронту сигнала C». То есть операция {CO,acc}={CO,acc}+FO выполняется по каждому положительному фронту C. Фигурные скобки обозначают объединение сигналов с различными именами в шину. Объединение может находиться как слева, так и справа от знака «=» в операции присвоения.

В качестве простейшего модуля верхнего уровня (который можно исполнить на симуляторе) возьмем пример, использующийся при описании процедурных языков:

```
module hello_word;        // интерфейсные порты отсутствуют
initial
   $display(«HELLO, WORLD !!!»);  // вызов системной функции
endmodule
```

После запуска симулятора должно получиться приблизительно следующее:

```
Highest level modules:
hello_word
 HELLO, WORLD!!!
0 simulation events
```

В данном примере была использована системная функция $display, которая используется для печати либо форматированной строки (как функция printf из библиотеки stdio в языке C), либо своих аргументов (как writeln в Паскале). Все системные функции и функции, написанные пользователем и подключенные через PLI-интерфейс, начинаются со знака $.

Немного забегая вперед, следует сказать, что язык Verilog (так же, как и VHDL) изначально предназначался для моделирования и средства синтеза появились позже. Поэтому часть конструкций языка не поддерживается синтезом. Одной из таких конструкций является initial. Для того чтобы осуществить начальную инициализацию, в синтезируемой модели следует предусмотреть специальный сигнал сброса.

```
module NCO_syn (FC, CO, C, Rst);  // имя модуля и список портов
input FC, C, Rst;  // входы
output CO;// выход
// описание используемых сигналов
wire [3:0] FC;
wire C, Rst; // не обязательно, так как по умолчанию вход является однобитным проводом
reg [3:0] acc;
reg CO;
// описание поведения системы
always @(posedge C or posedge Rst)  // событие — фронт C или Rst
 if (Rst)
 {CO,acc}=5'b0;
 else
 {CO,acc}={CO, acc}+FO;
endmodule
```

В данном примере добавлена процедурная конструкция if. Как можно видеть, она подобна конструкции if в языке C, нулевое значение в скобках соответствует false, ненулевое — true. Подробно о процедурных конструкциях будет рассказано позже, но следует заметить, что если в скобках стоит выражение, имеющее после вычисления биты со значением z или x, то выполняется ветвь else.

Для проверки работоспособности модуля используются испытательные стенды (testbench). Это модуль верхнего уровня, в котором могут использоваться несинтезируемые конструкции (initial) и типы данных (event, time, real). Также испытательные стенды содержат системные функции для вывода информации ($display, $write, $monitor), записи файлов изменения сигналов (vcd — value change dump) для последующего анализа, исследования статистических свойств сигналов и т. п.

Предположим, что модуль NCO будет использоваться для генерации частот 33 и 40 МГц при тактовой частоте 100 МГц. Для этого в накопительный сумматор на вход FC (frequency code) следует подать 11 и 13 соответственно. При этом полученные частоты будут 34,375 МГц (ошибка 1,375 МГц) и 40,9125 МГц (ошибка 0,9125 МГц), что, предположим, и требуется. В модуле testbench будут использоваться временные задержки — конструкции вида #NN, где NN — время в наносекундах. Более подробно различные виды задержек (delay) будут рассмотрены ниже. Оба модуля (NCO и testbench) могут быть записаны либо в один, либо в разные файлы.

```
`timescale 1ns/10ps   //директива симулятора — установка шага времени (необязательно, так как 1ns/10ps — значения по умолчанию)
module testbench;
// объявление сигналов
reg clk, rst;
reg [3:0] fc1,fc2
wire f1, f2;
// объявление переменных
integer clk_cnt, f1_cnt, f2_cnt;
real ratio;
//построение иерархии
NCO_syn nco1(.Rst(rst), .C(clk), .FC(fc1), .CO(f1)); //подключение по имени
NCO_syn nco2(fc2, f2, clk, rst); //подключение по расположению
initial
 begin
 clk=0;
 rst=0;
 fc1=4'd11;
 fc2=4'd13;
 clk_cnt=0;
 f1_cnt=0;
 f2_cnt=0;
 end
always #5 clk=~clk;  // генератор тактовой частоты 100 МГц
// управление
initial
 #1 rst=1'b1;  // формирование сброса
 #2 rst=1'b0;
 #1200;  //время симуляции
 $display(«toggle: clk «, clk_cnt, « , f1 «, f1_cnt, « , f2», f2_cnt); //вывод результатов
 ratio=100.0*$itor(f1_cnt)/$itor(clk_cnt); // $itor — преобразование integer в real
 $write(«freqv @ clk=100MHz f1=%f», ratio);
 ratio=100.0*$itor(f2_cnt)/$itor(clk_cnt);
 $display(« f2=%f», ratio);
 $finish;    // завершение симуляции
end
// сбор статистики
always @(posedge clk)
        clk_cnt=clk_cnt+1;
always @(posedge f1)
        f1_cnt=f1_cnt+1;
always @(posedge f2)
        f2_cnt=f2_cnt+1;
//два метода индикации (выбрать один, чтобы не засорять выход)
// 1
// печатает при изменении одного из сигналов
//initial $monitor(«Time %t clk %b rst %b f1 %b f2 %b»,$time,clk,rst,f1,f2);
// 2
//печатает по срезу clk
always @(negedge clk) $write(«Time %t clk %b rst %b f1 %b f2 %b \n», $time, clk, rst, f1, f2);
endmodule.
```

Как можно видеть из приведенного примера, блоков initial и always может быть сколько угодно, исполнение происходит одновременно (о том, как работает симулятор, будет рассказано при рассмотрении операции присваивания). Порядок исполнения различных конструкций определяется только по времени исполнения.

Построение иерархии (подключение модулей) возможно двумя способами: по имени (указываются имена портов, использованные при описании модуля) или по расположению (порядок сигналов такой же, как в описании модуля).

После запуска со вторым вариантом печати будет выдано следующее:

```
Compiling source file «testbench.v»
Compiling source file «nco.v»
Highest level modules:
testbench
Time 1000 clk 0 rst 0 f1 0 f2 0
Time 2000 clk 0 rst 0 f1 1 f2 1
Time 3000 clk 0 rst 0 f1 0 f2 0
Time 4000 clk 0 rst 0 f1 0 f2 1
Time 5000 clk 0 rst 0 f1 1 f2 0
Time 6000 clk 0 rst 0 f1 0 f2 0
Time 7000 clk 0 rst 0 f1 0 f2 1
Time 8000 clk 0 rst 0 f1 1 f2 0
Time 9000 clk 0 rst 0 f1 0 f2 1
[…]
Time 118000 clk 0 rst 0 f1 1 f2 1
Time 119000 clk 0 rst 0 f1 1 f2 0
Time 120000 clk 0 rst 0 f1 0 f2 1
toggle: clk 120 , f1 41 , f2 49
freqv @ clk=100MHz f1=34.166667 f2=40.833333
L41 «testbench.v»: $finish at simulation time 120300
```

Время измеряется в единицах, заданных вторым параметром директивы timescale (10 ps), и, соответственно, time 7000 означает 70 нс после старта симуляции. Общее время симуляции составило 1+2+1200=1203 нс. Размерность временных параметров, задаваемых в исходном коде модуля, определяется первым параметром директивы timescale (1 нс).

Вычисленные значения f1 и f2 отличаются от полученных моделированием. При увеличении длины выборки (времени моделирования) разница будет уменьшаться.

Продолжение следует