Проектирование для ПЛИС Xilinx на языке System Verilog в САПР Vivado

Илья ТАРАСОВ,
д. т. н.
ilya_e_tarasov@mail.ru

Сегодня актуализация языка System Verilog объясняется постоянным увеличением логического объема ПЛИС и связанной с этим необходимостью повышения производительности труда разработчиков. Язык System Verilog ориентирован в большей степени на моделирование сложных систем и создание комплексных автоматизированных тестов, что сокращает время на моделирование и отладку проектов. Появление поддержки System Verilog для синтеза в САПР Vivado позволяет начать практическое использование этого языка в проектах на базе ПЛИС Xilinx. Статья рассматривает базовые возможности System Verilog с учетом особенностей его поддержки в новой версии САПР Vivado.

Введение

Язык System Verilog был разработан на базе языков SuperLog и OpenVera. Затем был принят стандарт IEEE 1800-2005, а актуальной версией на настоящий момент является IEEE 1800-2009. Предпосылками к разработке System Verilog стали необходимость повышения уровня абстрагирования при описании схем, а главное — увеличение сложности моделирования и верификации проектов. Поэтому возможности System Verilog по сравнению с его предшественником Verilog развивались в первую очередь в области верификации. Соотношение возможностей языков System Verilog и Verilog показано на рис. 1.

Поддержка System Verilog добавлена в САПР Xilinx. В текущих версиях Vivado можно использовать System Verilog для описания схем. Моделирование на System Verilog находится в стадии реализации, однако могут быть применены программные продукты других производителей (например, Questa Sim от Mentor Graphics). Несмотря на ограниченную поддержку в Vivado, налицо тенденция к распространению этого языка и внедрению его в практику разработки наряду с VHDL и Verilog. Поэтому представляется полезным изучение основных возможностей System Verilog, чтобы впоследствии осознанно принимать решения о необходимости и степени его использования в новых проектах.

Поддержка типов данных

В языке Verilog все типы данных относятся к одному из двух больших классов: цепи (net) и переменные (variable). Они различаются тем, что состояние цепей вычисляется и обновляется непрерывно (по мере изменения входных сигналов), а переменные вычисляются и обновляются только внутри процедурных блоков. System Verilog не добавляет ничего нового к этой классификации — его типы также относятся к одному из этих двух классов. Дополнения, вносимые System Verilog в систему типов, лежат в основном в сфере моделирования.

В Verilog все сигналы имеют четыре возможных состояния 0/1/x/z, где x относится к состоянию «не определено/неизвестно», а z обозначает состояние высокого импеданса и может иметь физическое представление в блоках ввода/вывода FPGA (но не в логических ячейках). System Verilog вводит новые типы данных, каждый разряд которых может иметь два состояния — 0/1. Это ускоряет моделирование, что особенно актуально в случаях, когда разработчику интересно поведение сложного вычислительного модуля, в котором не используется состояние высокого импеданса, а состояния «не определено» устранены корректной моделью. Можно выполнять назначение данных, представленных четырьмя состояниями, объектам, представленным двумя состояниями, — в этом случае состояния x и z преобразуются в 0. К стандартным типам с двумя состояниями относятся int, shortint, longint, bit и byte. Типы Verilog и System Verilog приведены в таблице 1.

Назначение типов с двумя состояниями можно понять из их преимуществ и ограничений. Моделирование с помощью таких типов выполняется быстрее, потребность в памяти вдвое меньше (что позволяет моделировать более длительные процессы), но подразумевается, что электрические конфликты и неопределенности успешно устранены на других этапах проектирования.

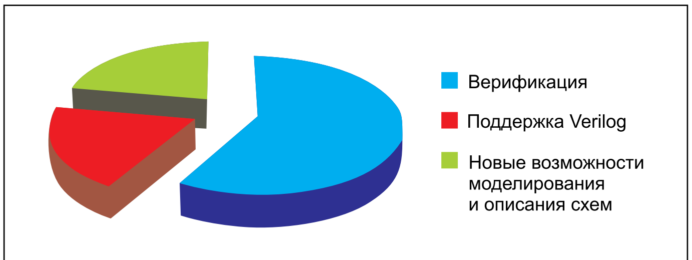

Рис. 1. Соотношение возможностей языков System Verilog и Verilog

Таблица 1. Типы данных Verilog и System Verilog

Тип	Число состояний	Знаковый/беззнаковый	Разрядность	Поддержка
shortint	2	Знаковый	16	System Verilog
int	2	Знаковый	32	System Verilog
longint	2	Знаковый	64	System Verilog
byte	2	Знаковый	8	System Verilog
bit	2	Беззнаковый	Определяется разработчиком	System Verilog
logic	4	Беззнаковый	Определяется разработчиком	System Verilog
reg	4	Беззнаковый	Определяется разработчиком	Verilog
integer	4	Знаковый	32	Verilog
time	4	Беззнаковый	64	Verilog

Переменная с четырьмя состояниями могла бы выявить эти конфликты, выведя x в соответствующих разрядах, но с двумя состояниями подобная ошибка будет замаскирована. Поэтому данные типы предназначены прежде всего для ускорения моделирования.

Тип logic подобен типу reg в Verilog. Однако reg может быть только переменной, тогда как logic может быть явно приведен к классу net. Например:

```
logic[15:0] data; // переменная
wire logic [15:0] wire_data; // цепь (net)
```

System Verilog допускает описание новых типов данных на основе существующих. Это может быть полезно в подобных случаях:

```
`ifdef FAST_MODEL
    typedef logic Tdata;
`else
    typedef bit Tdata;
`endif

module my_module(
input Tdata[7:0] input_data,
…
);
// тело модуля
endmodule;
```

В примере показано, как в зависимости от установленного (с помощью define) параметра FAST_MODEL тип Tdata определяется как logic или как bit. Описываемый далее модуль my_module будет использовать представления с 4 или 2 состояниями, что актуально при синтезе и моделировании соответственно. При этом переключение режима сводится к изменению строки `define FAST_MODEL, которая может управлять и другими модулями проекта.

Перечислимые (enumerated) типы задаются пользователем в виде списка всех возможных состояний:

```
enum reg[2:0] {IDLE = 0, S1 = 1, S2 = 2, S3 = 4} fsm;
```

Применение таких типов наглядно видно из приведенного примера, где перечислены состояния конечного автомата (FSM, Finite State Machine). Синтезаторы для FPGA эффективно распознают шаблоны конечных автоматов и используют оптимальный тип кодирования состояний. Однако для этого следует представить состояния не в виде жестко заданных констант (как это показано в примере, но только для того, чтобы продемонстрировать подобную возможность), а в виде перечня возможных состояний, для которых синтезатор способен выбрать номера. В примере принудительно задается нумерация по принципу one hot, что эффективно для КА с малым числом состояний.

Если принудительные значения в перечисляемом типе не заданы, используется нумерация по умолчанию — последовательно, начиная с 0.

Возможно описание групп состояний:

```
typedef enum bit[2:0] {IDLE, WORK[3], DONE} fsm_states;
```

В этом примере задается список состояний конечного автомата, содержащий кроме состояний IDLE и DONE набор состояний WORK0, WORK1, WORK2, WORK3.

В System Verilog применяется более строгий контроль типов, чем в Verilog, поэтому в ряде случаев требуется явное преобразование типов. Оно выполняется директивой $cast или оператором «’». Кроме этого, возможно описать приведение к знаковому/беззнаковому формату и приведение к определенной разрядности:

- **$cast(fsm_states, IDLE)** — возвращает код состояния IDLE из множества fsm_states;
- **signed’(a)** — возвращает знаковое представление переменной/сигнала a;
- **16’(5)** — возвращает константу 5, записанную в 16 двоичных разрядах.

Для удобства работы с данными добавлена поддержка структур (**struct**), объединений (**union**) и массивов (**array**). Эти понятия во многом похожи на свои аналоги в языке программирования С.

Пример структуры:

```
struct {
    int a;
    byte b;
} my_struct;
// пример использования
my_struct.a = 1;
my_struct = {1, 8`b10};
```

Описание структуры действительно очень близко к синтаксису языка С (используется ключевое слово **struct**). При этом в синтезируемом коде не создается каких-то специальных связанных между собой схем, а эффект от использования структур заключается в повышении читаемости кода и его более удобной организации. Структуры схожи с массивами, однако могут содержать объекты разных типов, тогда как массив представляет собой набор объектов одного и того же типа.

По умолчанию структуры не упакованы. Это означает, что отдельные элементы находятся в различных ячейках памяти. Такой подход ускоряет моделирование и упрощает доступ, однако в памяти появляются пропуски. Для упакованных структур все данные, относящиеся к структуре, хранятся в виде непрерывной последовательности. Упаковка структур задается ключевым словом packed:

```
packed struct {
            int a;
            byte b;
} my_packed_struct;
```

Поскольку на байт отводится 8 бит, а на int — 32, такая структура будет храниться в заранее прогнозируемом виде. К элементу a можно обратиться не только по его явному имени, но и как my_paked_struct[39:8] (элемент b займет соответственно my_packed_struct[7:0]).

Кроме этого, можно явно указать, будут элементы упакованной структуры беззнаковыми (это подразумевается по умолчанию) или знаковыми.

```
struct packed unsigned {
    int a;
    shortint b;
} my_unsigned_packed_struct;

struct packed signed {
    int a_signed;
    shortint b_signed;
} my_unsigned_packed_struct;
```

Кроме структур, в System Verilog добавлены и объединения (**union**). Они имеют похожее объявление, но отличаются тем, что все элементы, входящие в объединение, разделяют общую область памяти:

```
typedef union packed {
    bit[31:0] data;
    bit[3:0][7:0] data_array;
} union_type;

union_type sample_data;

// sample_data.data[23:16] относится к тем же данным, что и
sample_data.data_array[2]
```

Объединения можно пометить ключевым словом **tagged**. Добавление тэга (tag) означает, что вместе с объединением будет храниться признак, позволяющий узнать, какой именно из перечисленных типов данных был записан в объединение последним:

```
typedef union tagged {
    int a;
    ogic[31:0] b;
} tagged_union;

tagged_data data; // объявление тэгированных данных
int data_out;

data = tagged a 7; // установка тэга, чтобы отразить факт записи в поле a
data_out = data.a; // корректно
data_out = data.b; // неверно! Тэг не совпадает с типом данных,
к которому делается попытка доступа
```

Так же, как и Verilog, язык System Verilog обеспечивает поддержку массивов (array). По сравнению с Verilog имеется ряд улучшений.

В стандарте Verilog-95 массивы могли иметь до двух измерений, причем одно из них является упакованным, а другое неупакованным. Verilog-2001 допускает уже несколько измерений с неупакованными данными:

```
reg[7:0] num [1023:0][7:0].
```

System Verilog допускает несколько измерений упакованных данных. Элементами массивов могут быть также структуры и объединения. Добавлены и несинтезируемые конструкции (используемые только для моделирования) — динамические массивы, ассоциативные массивы и очереди.

Для доступа к параметрам массивов удобно использовать встроенные функции:

- **$left** — возвращает левый индекс;
- **$right** — возвращает правый индекс;
- **$low** — возвращает наименьший индекс;
- **$high** — возвращает наибольший индекс;
- **$increment** — возвращает «1», если **$left** больше или равно **$right**, и «–1» в противном случае;
- **$size** — возвращает размер массива;
- **$dimensions** — возвращает количество измерений массива;
- **$unpacked_simensions** — возвращает количество неупакованных измерений массива.

Перечисленные функции могут быть весьма полезны при разработке модулей, разрядность данных в которых подвержена изменениям в процессе отладки и уточнения функциональности. Например, если разработчик, используя 8-битные данные, опрашивал значение старшего разряда как **data**[**7**], то при переходе к 16-битным данным ему придется вспомнить, что **data**[**7**] больше не является старшим разрядом. В то же время выражение **$high**(**data**) в любом случае вернет индекс старшего разряда. Аналогичное назначение имеют и остальные функции. Следует стремиться к тому, чтобы описывать доступ к данным, чья разрядность может измениться, в максимально общем виде.

Следующие понятия являются несинтезируемыми и могут быть использованы только при моделировании.

Динамические массивы:

```
integer dyn_array1[]; // описание пустого динамического массива
dyn_array1 = new[5](`{1, 2, 3, 4}); // заданы первые элементы массива
dyn_array1 = new[8](dyn_array1); // изменяется размер с сохранением предыдущих элементов
```

Вызов функции **size**() возвращает текущий размер массива. Динамический массив можно удалить деструктором **delete**().

Очереди (queue) используются для описания массивов с автоматически изменяемым размером при добавлении и удалении данных. Очереди описываются как неупакованные массивы, но используют символ $ вместо размера массива. Этот символ также указывает на конец очереди:

```
integer a[$]; // описание очереди, размер не указывается
int b[$] = {1, 2, 3}; // очередь с начальным состоянием
logic c[$:15]; // очередь с максимальным размером 16 бит

a[0] = 10; // записать 10 в начало очереди a
b = {b, 4}; // добавить 4 в конец очереди b
a = {}; // очистить очередь a
```

Можно еще раз подчеркнуть, что динамические массивы и очереди применяются только для моделирования. Они не могут быть синтезированы, поскольку это подразумевает наличие в ПЛИС памяти заранее неизвестного объема.

Операторы

Основной набор операторов System Verilog совпадает с операторами Verilog (и схож с операторами языка C). Для удобства разработчика добавлены некоторые операторы, упрощающие запись арифметических и логических выражений. Операторы являются синтезируемыми, то есть могут описывать не только тесты, но и схему в ПЛИС.

Операторы присваивания

Операторы присваивания += –= *=/=%=?= |= ^= аналогичны используемым в языке C. Такой формат не поддерживается в Verilog. Очевидно, что эффект от этих операторов может быть достигнут и иначе — например, A += B означает то же самое, что A = A + B.

Также операторы >>= <<= используются для логического сдвига, а >>>= <<<= для арифметического. Добавлены и известные в C унарные операторы ++ и –. Они обозначают инкремент и декремент соответственно.

Операторы проверки

Операторы проверки с учетом символа подстановки (wildcards) используются для проверки величин, принимающих четыре состояния. Разряды, имеющие значения z или x, успешно проходят сравнение с любым значением 0/1/z/x.

A ==? B — проверяет равенство A и B.

A !=? B — проверяет неравенство A и B.

Оператор inside

Оператор inside проверяет попадание величины в указанный список. Результат равен 1, если проверяемая величина присутствует в списке:

```
a inside {[1:8]} // проверка на попадание в интервал
a inside {1, 2, 4, 8, 16} // проверка на равенство одному из перечисленных значений
```

Потоковые операторы

Потоковые операторы >> и << осуществляют упаковку данных в последовательность в заданном формате:

```
int x = {"A", "B", "C", "D"};
{>>{X}} // генерирует "A" "B" "C" "D"
{<< byte {X}} // генерирует "D" "C" "B" "A"
{<< 16 {X}} // генерирует "C" "D" "A" "B"
```

Структуры управления

Синтезируемые процедурные блоки

В языке Verilog основным процедурным блоком является **always**. С его помощью можно описывать как комбинаторную логику, так и синхронные схемы. Возможности этого оператора расширены в System Verilog, где добавлены три его разновидности:

- **always_comb** используется для описания комбинаторной логики;
- **always_ff** описывает синхронную (тактируемую) логику;
- **always_latch** предназначен для описания защелок (однако Xilinx не рекомендует применять защелки в FPGA).

Если сигналам присваиваются значения внутри одного из таких блоков, то этим сигналам уже не могут быть присвоены значения вне этого блока. Данное ограничение не касается обычного блока **always**. Использование дополнительных блоков при разработке дает ряд преимуществ:

- синтезатор может провести дополнительные проверки, например предупредить разработчика, что для блока **always_ff** в действительности не оказалось реализовано ни одного тактируемого компонента;
- упрощается построение синтезаторов и средств моделирования, ускоряется их работа;
- повышается читаемость и выразительность кода, облегчается его поддержка и отладка.

Процедурные блоки для моделирования

В Verilog существуют блоки **initial** и **always**, предназначенные для описания внешних воздействий, происходящих в начале моделирования (**initial**) и циклически в процессе моделирования (**always**). System Verilog добавляет к ним блок final, выполняемый при завершении моделирования. Его применение выглядит следующим образом:

```
final
begin
    // операторы, например вывод сообщения в консоль
    $display("Моделирование завершено");
end
```

Расширенные возможности циклов в System Verilog

Возможности цикла for, имеющегося в Verilog, были несколько расширены в System Verilog. Кроме этого, добавлено несколько новых вариантов циклов. Отличия между Verilog и System Verilog приведены в таблице 2.

Цикл **do … while** не поддерживается в Verilog. Он выглядит следующим образом:

```
do begin
    // операторы
end while(<условие>)
```

Тело цикла выполняется по крайней мере один раз. После ключевого слова **while** записывается условие выхода из цикла.

Цикл **foreach** (итератор) введен для упрощения описания действий с массивами. Итератор позволяет не указывать точный размер массива, а определяет его самостоятельно, например:

```
foreach(x[i])
    // действие для очередного элемента массива x[]
```

Для многомерных массивов можно указывать несколько переменных-счетчиков. Можно также не указывать переменные, для которых не требуется циклическое исполнение:

```
int a[10][20][30]
foreach(i, , k)
// итерации будут повторяться для первого (0..9) и третьего (0..29)
измерений массива
```

Операторы передачи управления

Операторы передачи управления расширяют возможности описания управляющих структур. К этим операторам относятся:

- **break** — досрочное завершение цикла;
- **continue** — переход к концу итерации и проверка условия продолжения цикла;
- **return** — возврат из функции/задачи.

Управление шифратором приоритета

Управление шифратором приоритета в операторе case позволяет описать непосредственно в синтезируемом коде особенности поведения схемы в том случае, если возможно выполнение более чем одного условия. Ниже приведен соответствующий пример. В данном случае конкретные действия не имеют решающего значения, важнее то, что отдельные условия не перекрываются, и в любом случае будет выполнен строго один оператор:

```
always @(*)
  case(operation)
    2'b00: x = a + b;
    2'b01: x = a – b;
    2'b10: x = a | b;
    2'b11: x = a & b;
endcase
```

Модификатор **priority** указывает, что возможно выполнение более чем одного условия, поэтому важно учитывать порядок их описания. Первое из описанных условий будет выполнено. В качестве примера приведен оператор case, в котором условия закодированы по правилу one-hot, то есть каждый разряд соответствует признаку «операция выбрана». При этом возможна ситуация, когда будут установлены два и более разрядов. С ключевым словом **priority** будет выполнена первая из строк, для которой найден установленный разряд:

```
always @(*)
  priority case(operation)
    4'b???1: x = a + b;
    4'b??1?: x = a – b;
    4'b?1??: x = a | b;
    4'b1???: x = a & b;
endcase
```

Модификатор **unique** указывает, что возможно выполнение только одного из условий (по соображениям, известным разработчику). При этом выполнение одного из условий обязательно. Модификатор **unique0** аналогичен unique, но допускается отсутствие совпадений:

Таблица 2. Различия в организации цикла for в языках Verilog и System Verilog

Verilog	System Verilog
Переменная-счетчик в Verilog должна быть объявлена до ее использования в цикле: reg [15:0] i; **for** (i = 0; i <= 15; i = i + 1)	В System Verilog объявление переменной-счетчика допустимо внутри цикла: **for** (reg [15:0] i = 0; i <= 15; i = i + 1)
Вложенные циклы должны использовать собственные переменные-счетчики для каждого уровня вложенности	На каждом уровне вложенности можно использовать одно и то же имя переменной-счетчика, которая будет определена локально на данном уровне цикла
Автоматических переменных не предусмотрено	Переменные в цикле являются автоматическими — они создаются при входе и уничтожаются при выходе из цикла
Множественные присваивания не поддерживаются	Поддерживаются множественные присваивания при инициализации цикла и на шаге цикла: for (int cnt=1, j=0, done=0; cnt*j < 128; cnt++, j++)

```
always @(*)
  unique case(operation)
    4'b???1: x = a + b;
    4'b??1?: x = a – b;
    4'b?1??: x = a | b;
    4'b1???: x = a & b;
endcase
```

Функции, задачи и пакеты

Функции и задачи также несколько расширены в System Verilog. Дополнительно добавлена возможность описания пакетов (**package**), что приближает System Verilog к возможностям структурирования проектов, имеющимся в языке VHDL.

Функции (**function**) и задачи (**task**) призваны обеспечить удобный вызов многократно исполняемого кода. Они в чем-то схожи с подпрограммами в языках программирования, а разделение на задачи и функции связано с их основными особенностями. Например, все операторы функции выполняются в течение одного временного слота, а задача может включать в себя операторы wait или # (то есть #10 описывает в задаче задержку на 10 единиц времени, а внутри функции такой оператор не может быть использован). Кроме того, функция не может вызывать задачи, но задача может вызывать функции и другие задачи.

Функция, как и в языках программирования, должна вернуть единственное значение и может быть использована в качестве элемента выражения. Задача не возвращает значения и не может быть использована в выражениях.

В System Verilog для функций и задач введен ряд расширений. В частности, необязательны «операторные скобки» **begin … end**, поддерживаются операторы управления исполнением **break**, **continue**, **return** <value>. Кроме того, функции в System Verilog могут и не возвращать значение (иметь тип void), не иметь ни одного аргумента или иметь аргументы с направлением передачи out, inout. В качестве аргументов допустимы структуры и объединения.

По умолчанию функции и задачи являются статическими (**static**). Такой тип аналогичен обычным функциям в Verilog. Однако кроме статических, существуют автоматические (**automatic**) и постоянные (**constant**) функции.

Автоматические функции создаются в момент их вызова, и требуемые ими программные ресурсы освобождаются в момент их завершения. Речь идет, конечно же, о фазе моделирования. Каждый экземпляр автоматической функции имеет собственный набор ресурсов, что позволяет, например, выполнять рекурсивные вызовы. Ниже приведен текст с описанием функции, вычисляющей факториал, которая представляет собой широко распространенный пример, демонстрирующий возможности рекурсии (хотя в реальных проектах вычисление факториала удобнее заменять таблицей значений):

```
module demo_automatic;

function automatic integer factorial (input[31:0] operand);
integer i;

if (operand >= 2)
  factorial = factorial(operand-1)*operand;
else
  factorial = 1;
endfunction
```

Функции типа **constant** вычисляются только на этапе elaboration (он предшествует как синтезу, так и моделированию). Приведенный ниже листинг демонстрирует пример такой функции. Она не синтезирует схему, а требуется для более удобного задания размеров памяти. В процессе elaboration значение этой функции будет вычислено и подставлено в качестве константы в RTL-описание:

```
function integer log2i(input[31:0] value);
begin
    value = value – 1;
  for(log2i = 0; value > 0; log2i = log2i + 1)
    value = value >> 1;
end
endfunction
```

Еще одним дополнением является возможность передачи аргументов в функцию по ссылке (pass by reference). В данном случае в функцию передается не текущее значение аргумента, а ссылка (адрес) этого аргумента. Приведенный ниже пример показывает практические различия при передаче аргумента по значению (pass by value) и по ссылке:

```
module demo_args;
int a;
initial
  begin
  #10 a = 10;
  #10 a = 20;
  #10 a = 30;
  #10 $finish;
  end
```

```
task automatic pass_by_val(int x);
  forever
    @x $display("pass_by_val: X is %0d", x);
  endtask

  task pass_by_ref(ref int x);
  forever
    @x $display("pass_by_ref: X is %0d", x);
  endtask

initial
  pass_by_val(a)
initial
  pass_by_ref(a);
endmodule
```

Выводом этого модуля будет:

- pass_by_val: X is 10;
- pass_by_ref: X is 10;
- pass_by_ref: X is 20;
- pass_by_ref: X is 30.

Можно увидеть, что аргумент, переданный по значению, оказался напечатан только один раз вследствие того, что в процедурном блоке **initial** был произведен вызов **pass_by_val(a)**. Находящаяся внутри функции конструкция @x неспособна выявить изменения аргумента, поскольку каждый раз функция получает его текущее значение, а не «историю изменений». Напротив, при передаче в функцию ссылки на сигнал a все изменения сигнала приводят к срабатыванию @x, и каждое присваивание нового значения приводит к вызову функции **pass_by_ref**.

Пакеты

Пакеты (**packages**) знакомы разработчикам на VHDL и введены в System Verilog с теми же целями — создание в проекте коллекций констант, типов, функций, задач, параметров и т.п., разделенных по областям видимости. Эффект от применения пакетов можно легко сравнить с подключением файлов директивой **include**, которая добавляет в проект все содержимое указанного файла:

```
package ComplexPkg;
typedef struct {
  float i, r;
} Complex;

function Complex add(Complex a, b);
  add.r = a.r + b.r;
  add.i = a.i + b.i;
endfunction

function Complex mul(Complex a, b)
  mul.r = (a.r * b.r) — (a.i * b.i);
  mul.i = (a.r * b.i) + (a.i * b.r);
endfunction

endpackage : ComplexPkg
```

Можно непосредственно указать на расположение описанной в пакете функции:

```
ComplexPkg::Complex cout = ComplexPkg::mul(a, b);
```

Описанные в пакете объекты можно также импортировать в другой модуль:

```
import ComplexPkg::Complex;
import ComplexPkg::mul;
```

Интерфейсы

При разработке сложных устройств часто используется передача с помощью групп сигналов или шин. Например, тактовый сигнал, шина адреса, шина данных и сигналы управления могут образовывать системную шину процессорной системы. При этом для ведущего и ведомого устройств (master и slave) направление некоторых сигналов может отличаться. Понятие интерфейса служит в System Verilog для упрощения описания таких систем.

Пример описания интерфейса:

```
interface mem_bus;
  logic [31:0] write_data;
  logic write;
  logic [31:0] addr;
endinterface
```

Этот интерфейс в дальнейшем можно использовать в модулях:

```
module mem_controller(
  mem_bus bus; // используется ранее объявленный интерфейс
  input clk; // другие сигналы
);
```

В системах с ведущим и ведомым устройствами направления сигналов могут различаться. Указать эти направления можно с помощью модификаторов (modports), например:

```
modport master(
  input clk, data_read,
  output data_write, addr, write_enable
);

modport slave(
  input clk, data_write, addr, write_enable,
  output data_read
);
```

В показанном примере ведущее устройство является источником адреса, данных для записи и сигнала разрешения записи. В свою очередь, ведомое устройство принимает эти сигналы и возвращает в ведущее устройство читаемые данные. Указать нужную модификацию можно в формате **<interface>.<modport>** — например, **mem_bus.master**.

Моделирование

В начале статьи было отмечено, что повышение эффективности моделирования было основной причиной разработки System Verilog. Поэтому в языке можно наблюдать разносторонние (и не обязательно используемые одновременно) возможности описания тестовых последовательностей, структурного и объектно-ориентированного программирования и т.п.

Классы

Классы (**class**) являются широко известным элементом объектно-ориентированного программирования. Их применение призвано решить целый ряд задач по разграничению области видимости, автоматическому управлению выделением памяти и т.д. Классы в System Verilog могут быть использованы для описания тестовых воздействий и моделирования. Объявление класса начинается с ключевого слова class, за которым внутри «скобок» **class/endclass** описываются компоненты этого класса — переменные, функции и т.д.

```
class MyDataClass;
  real x;
  real y;
  int a;

  function MyDataClass_Function1(input real a);
    // …
  endfunction

endclass
```

Программы

Программы (**program**) предназначены для структурирования тестов путем обобщения функционально связанных процедурных блоков. Программа **может** содержать: объявления данных, классов, подпрограмм и один или несколько процедурных блоков **initial/final**. Программа **не может** содержать: процедуры **always**, подключаемые модули и интерфейсы.

Пример программы:

```
program init_some_vars(input a, b, c);
initial begin
  a = 0;
  b = 0;
  c = 0;
end
endprogram
```

Эту программу впоследствии можно вызывать из модуля:

```
module testmodule;
// …
init_some_vars init_abc(c, d, e);
```

В описании программы используется установка в 0 переменных, описанных как формальные параметры. Однако внутри модуля в программу передаются фактические параметры c, d, e, которые и будут обнулены. Программы могут быть полезны, если в тестовом модуле требуется описывать сложные действия, применяемые к различным переменным.

Блок синхронизации

Блок синхронизации (**clocking block**) служит для упрощения описания соотношения сигналов в тактируемых компонентах. Упрощенные поведенческие модели предполагают, что сигналы изменяются мгновенно и так же мгновенно становятся доступными для считывания последующими компонентами в цепи. Это существенно ускоряет моделирование, однако при этом программы моделирования полагаются на то, что разработчик уже обеспечил требуемые времена распространения сигналов и они окажутся в нужный момент на соответствующих входах. Однако в действительности данные, записываемые в триггер, должны присутствовать на входе в течение времени t_{setup} (время установки) до фронта тактового сигнала и удерживаться в течение времени t_{hold} (время удержания) после фронта (рис. 2). Моделирующие программы могут учитывать эти времена только в режиме Post-Route, когда работают с физическими моделями компонентов на кристалле. Но в этом режиме время моделирования существенно увеличивается, и общим подходом является искусственное указание времен распространения сигналов в более быстром поведенческом (behavioral) режиме моделирования. Это не означает, что САПР впоследствии будет стараться обеспечить именно такие времена распространения при синтезе схемы, однако поведенческий режим с искусственно введенными задержками дает все же более адекватную картину, чем моделирование без задержек вообще. Блок синхронизации является примером конструкции System Verilog, упрощающей и унифицирующей такое описание применительно к тактируемым компонентам.

Блок синхронизации имеет следующий синтаксис:

```
clocking master_cb @(posedge clk)
  default input #1step output #1ns; // задержки по умолчанию
  input #2 d; // задержка отличается от установленной по умолчанию
endclocking
```

Конструкция fork-join

Конструкция **fork-join** используется для синхронизации нескольких процедурных блоков при моделировании. Ее работу удобнее рассмотреть на практическом примере. Предположим, что в блоке описано несколько событий, происходящих в разное время.

```
initial begin:
fork
  begin
    #2 $display("Task 1");
  end
  begin
    #1 $display("Task 2");
  end
    #3 $display("Task 3");
  join
    #1 $display("Common task");
end
```

Выводом данной модели будет:

- **Task2**;
- **Task1**;
- **Task3**;
- **Common task**.

Однако, рассматривая только времена, которые установлены для функции **$display**, можно заметить, что вывод должен был бы происходить в другом порядке — сначала должны сработать функции, которым установлена задержка в 1 единицу времени модели. Например, **Common task** должно выполниться раньше, чем **Task1** и **Task3**. Однако порядок выполнения определяется с учетом конструкции **fork-join**. В данном случае **fork** отмечает начало нескольких параллельно выполняющихся процессов, а **join** заставляет программу моделирования дождаться завершения всех

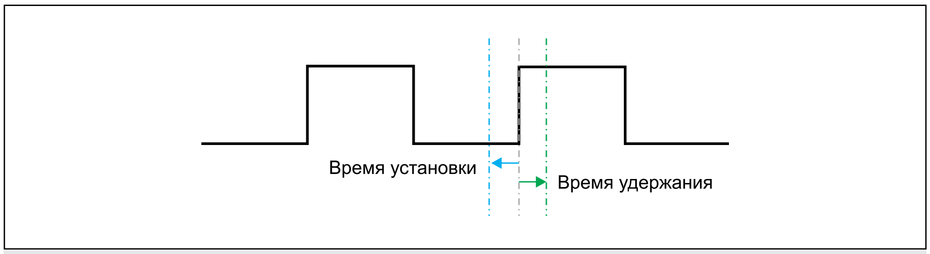

Рис. 2. Время установки и время удержания в тактируемых компонентах

этих процессов. В итоге оператор задержки #1 в строке **$display("Common task")** будет отсчитывать время не от начала моделирования, а от срабатывания оператора **join**.

Другая разновидность, оператор **join_any**, продолжит выполнение модели при срабатывании любого из вложенных в fork процессов. Наконец, **join_none** определяет, что дожидаться завершения запущенных процессов не следует, а остальная часть процедурного блока может быть запущена сразу же.

Генерация случайных воздействий

Усложнение проектов на базе цифровых микросхем делает все более актуальным их тестирование в процессе разработки. Для систем связи или цифровой обработки сигналов невозможно выполнить полное покрытие тестами из-за огромного количества сочетаний входных воздействий. Практичнее выполнять тесты, сочетающие проверки распространенных фиксированных сценариев и контролируемые случайные воздействия (constrained random simulation). Контроль случайных воздействий нужен для того, чтобы сконцентрироваться на проверке наиболее вероятных сценариев — например, при передаче текстовых файлов в формате ASCII вероятнее появление в канале связи байтов, соответствующих буквам, цифрам и знакам препинания, а не всех символов набора ASCII в одинаковых пропорциях. Поэтому разработчику было бы удобно задавать желаемые законы распределения случайных значений.

В Verilog существует функция **$random()**, возвращающая случайное (псевдослучайное) значение. Разница между случайным и псевдослучайным заключается в способе их получения. Строго говоря, случайное значение должно быть получено из источника, который физически способен его сформировать, — например, генератор на тепловых шумах. Однако программные генераторы случайных чисел существенно проще в реализации и повсеместно распространены. Они формируют последовательности, которые на самом деле являются детерминированными, но регулярность в них не прослеживается на первый взгляд, поэтому на практике они могут служить рабочим инструментом для тестирования цифровых систем. Детерминированность псевдослучайных последовательностей полезна тем, что при одинаковых начальных условиях они повторяются, а потому становится возможным воспроизводить какой-либо редкий эффект, возникающий только при определенном сочетании входных воздействий.

В дополнение к **$random()** **System Verilog** добавляет функции **$urandom()** и **$random_range()**. Функция **$urandom** (unsigned random) возвращает псевдослучайное число без знака (в диапазоне 0 .. 2^{32}–1). Функция **$random_range** (max, min) возвращает псевдослучайное число в диапазоне min..max. Генератор случайных чисел может быть настроен вызовом функции **srandom()**, которая принимает в качестве аргумента целое число.

Функция **randomize()** служит для задания случайных значений отдельным объектам. Она входит в состав пакета std. Например, следующий вызов установит две переменные в случайные значения, которые будут укладываться в допустимый для их типа диапазон:

```
reg [3:0] r1;
reg [3:0] r2;
initial begin
  std::randomize(r1, r2);
end
```

Для этой функции хорошо проявляются дополнительные возможности System Verilog. В частности, с помощью конструкции **randomize .. with** можно получать достаточно мощные эффекты:

```
reg [3:0] r1;
reg [3:0] r2;
initial begin
  std::randomize(r1, r2) with {
    r1 < r2;
    r1 + r2 == 4;
  };
end
```

Конструкция **with** задает набор условий, которые должны выполняться для псевдослучайных значений. Для описанных в примере условий подходящим вариантом является r1 = 1, r2 = 3. Причем теоретически возможно задать такой перечень условий, который не сможет быть выполнен. В этом случае будет сформирована ошибка.

Конструкция randcase

Конструкция **randcase** служит для формирования псевдослучайных значений с заданными весами. В приведенном ниже примере переменная x принимает значения 1, 2 или 3, причем частота их появления пропорциональна весам, указанным в конструкции **randcase** (в примере это 3, 1 и 4 соответственно):

```
randcase
   3 : x = 1;
   1 : x = 2;
   4 : x = 3;
endcase
```

В качестве весов можно использовать не только числа, но и выражения и даже функции:

```
a = 1; b = 2;
randcase
   a + b : x = 1;
   a * b : x = 2;
   4 : x = 3;
endcase
```

Случайные последовательности randsequence()

Случайные последовательности **randsequence**() удобны для моделирования конечных автоматов. В примере используется автомат, изначально находящийся в состоянии s0. Из этого состояния он может сформировать последовательность s1 → s2 → s3. В свою очередь, из s1 возможно попасть в состояния s1a, s1b, а из s2 — в s2a, s2b:

```
initial
   randsequence (s0)
      s0 : s1 s2 s3;
      s1 : s1a | s1b;
   s2 : s1a | s2b;
   endsequence
```

Для переходов можно задать весовые коэффициенты, регулирующие вероятность их возникновения:

```
s1 : s1a := 1 | s1b := 9;
```

Модификаторы rand и randc

Модификаторы **rand** и **randc** используются совместно с переменными для указания на то, что они являются случайными. Отличия этих модификаторов в том, что **randc** указывает на цикличность (*cyclic*, откуда в его название добавлен символ «c») значений, то есть при их присваивании значения не будут повторяться, пока не примут все возможные состояния из допустимого для данной переменной предела.

Пример:

```
rand bit[3:0] a;
```

Модификаторы можно дополнять ограничениями:

```
randc bit[3:0] a;

constraint num_constr {
   a > 10;
   a < 15;
}
```

Ограничения, кроме обычных условий, могут описывать и распределения вероятности появления отдельных значений:

```
constraint constr_interval {
   a dist {
      [1:5] := 10;
      [6:10] := 5;
      [11:20] := 2;
   }
}
```

В данном примере после ключевого слова **dist** заданы интервалы значений для переменной *a* и весовые коэффициенты для вычисления вероятности попадания в эти интервалы. Вероятность каждого интервала определится как частное от весового коэффициента и суммы всех весовых коэффициентов. Кроме коэффициента для интервала, можно также определить коэффициент для каждого значения из этого интервала. В этом случае вероятность будет тем выше, чем больше значений попадает в каждый диапазон:

```
constraint constr_interval {
   a dist {
      [1:5] := 10;
      [6:10] := 5;
      [11:20] := 2;
   }
}
```

Условные ограничения

Условные ограничения определяют соотношения между выполнением одного из условий и наличием ограничения для другой величины. Например, если объявлены случайные величины *a* и *b*, можно задавать соотношения между ними при помощи оператора импликации ->, как показано в примере:

```
constraint impl1 {
   a == 1 -> b < 5;
}
```

Это объявление означает, что если выполняется условие «a равно 1», то для b будет назначено случайное значение, меньшее 5.

С оператором импликации связан оператор **solve**, управляющий порядком вычисления ограничений:

```
constraint constr1 { a -> b == 0;}
constraint c_order {solve a before b;}
```

Данный пример задает соотношения между *a* и *b* таким образом, что, как только *a* примет значение «истина», *b* должно принимать значение 0. Приведенный ниже пример явно задает иной порядок действий — сначала определяется случайное значение для переменной *b*, и только если она равна 0, становится возможным задание для *a* ненулевого значения:

```
constraint constr1 { a -> b == 0;}
constraint c_order {solve b before a;}
```

Обеспечение покрытия тестами

Покрытие тестами является важным аспектом организации тестирования сложных проектов. Часто оказывается, что перебрать все возможные сочетания входных параметров нельзя по объективным причинам ввиду их огромного количества. Тестирование проекта с заданием всех возможных значений занимало бы время, не просто нецелесообразное с точки зрения сроков выполнения проекта, но и технически недостижимое (например, если речь идет о числе сочетаний, определяемых факториальными выражениями). Поэтому весьма важным является составление плана верификации с перечислением конкретных функций, которые должны быть проверены. Количественной характеристикой выполнения этого плана является процент проведенных тестов. Эта величина называется покрытием тестами (coverage). Формирование тестов, которые должны быть проведены, очень трудно поддается автоматизации, поскольку при этом легко получить большое число тестов, проверяющих второстепенные функции или многократно тестирующих узлы проекта, для которых достаточно однократной проверки. Поэтому возможности System Verilog по обеспечению покрытия тестами должны сочетаться с грамотной организационной работой.

Базовой конструкцией здесь является понятие **covergroup**:

```
covergroup cg;
…
endgroup

cg cg_inst = new;
```

В примере описана группа cg и создан ее экземпляр **cg_inst** с помощью конструктора **new**. Группа содержит одну или более точек покрытия (coverage point). С ними ассоциированы «корзины» (bins), которые представляют собой счетчики событий, определяемые пользователем. Увеличение этих счетчиков происходит либо явным образом путем выполнения метода **sample**(), либо неявно, при наступления события, определенного в **covergroup**. Например, следующая строка описывает подсчет изменений переменной **data**:

```
always @(data) cg_inst.sample();
```

Корзины могут быть определены дополнительно. Приведенный ниже пример показывает определение дополнительных корзин для подсчета числа назначений переменной **my_var** значений в диапазоне 0–10 и 7:

```
covergroup var_cg @(my_var);
   cov_point : coverpoint my_var {
      bins b_0_10 = {[0:10]};
      bins b_7 = {7};
      bins others = default;
   }
endgroup

var_cg my_var_cg;
initial my_var_cg = new;
```

Таблица 3. Параметры корзин

Имя параметра (тип)	Значение по умолчанию	Описание
weight (number)	1	Весовой коэффициент служит для вычисления реального покрытия тестами при проведении моделирования. Количество попаданий в корзину умножается на этот коэффициент
goal (number)	90	Количество попаданий, требуемое для достижения цели покрытия тестами
name (string)	name	Имя, заданное пользователем (если не задано, генерируется автоматически)
comment (string)	“”	Комментарий
at_least (number)	1	Минимально требуемое количество попаданий в корзину
detect_overlap (boolean)	0	Если это свойство установлено в «истину», генерируется предупреждение, если одно условие попадает в две или более корзины
auto_bin_max (number)	64	Максимальное количество автоматически создаваемых корзин
cross_num_print_missing (number)	0	Количество пропущенных (не определенных) кросс-корзин, которые должны быть сохранены и выведены в отчете
per_instance (boolean)	0	При установке в «истину» статистика собирается по каждому экземпляру covergroup

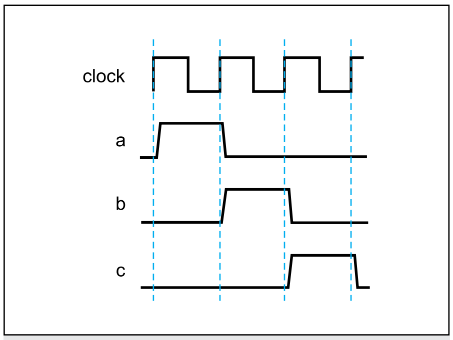

Рис. 3. Пример последовательности сигналов

Для корзин могут быть определены не только статические условия для подсчета количества состояний, но и последовательности. Например, если требуется проверить попадание конечного автомата последовательно в состояния S1, S2, S3, это можно определить следующим образом:

```
bins my_cycle = (S1 => S2 => S3);
```

Допустимы синтаксические выражения, упрощающие определение сложных и повторяющихся переходов. Например, (1, 5 => 6, 7) эквивалентно любым переходам из левой части к правой: (1 => 6), (1 => 7), (5 => 6), (5 => 7). Выражение 5[*3] соответствует трехкратному повторению значения 5 (5 => 5 => 5). Переменное число повторений задается следующим образом: 3[*2:4] эквивалентно любому из этих списков (3=>3), (3=>3=>3), (3=>3=>3=>3). Символ -> определяет списки, в которых второе значение может появляться не сразу же после первого: 3[->2] эквивалентно (3=>…3).

Конструкция **ignore_bins** служит для задания значений, которые не должны подсчитываться:

```
covergroup cg;
  coverpoint a {
    ignore_bins ignore_vals = {7, 8};
    ignore_bins ignore_trans = (1 => 3 => 5);
  }
endgroup
```

Кроме игнорирования значений, их можно трактовать как ошибочные. В этом случае появление значений или последовательностей будет формировать ошибку в процессе моделирования:

```
covergroup cg;
  coverpoint a {
    illegal_bins ignore_vals = {7, 8};
    illegal_bins ignore_trans = (1 => 3 => 5);
  }
endgroup
```

Одновременное изменение двух переменных подсчитывается при указании ключевого слова **cross**:

```
covergroup cg @(posedge clk);
  ab : cross a, b;
endgroup
```

Для определяемых корзин можно устанавливать параметры. Их список приведен в таблице 3.

Свойства могут быть определены в исходном тексте на System Verilog:

```
covergroup cg;
  a : coverpoint a_var;
  b: coverpoint b_var;
endgroup

initial begin
  cg cg1 = new;
  cg1.option.comment = “This is a comment”;
end
```

После сбора статистики методом **sample()** или по результатам определенных в группе условий можно воспользоваться следующими встроенными методами для получения результатов:

- **get_coverage()** — возвращает значение покрытия тестами для данного типа;
- **get_inst_coverage()** — возвращает значение покрытия тестами для отдельного экземпляра.

Метод **start()** начинает сбор статистики, а метод **stop()** приостанавливает его.

Проверка утверждений

Утверждения (assertions) являются основным инструментом верификации. Они располагаются в синтезируемом коде и описывают логические выражения, которые **должны выполняться**. Так, следующий фрагмент кода проверяет, что некий управляющий автомат находится в состоянии «запрос» (request), когда у него имеется по крайней мере один источник сигнала req:

```
always @(posedge clk)
  if (state == REQ)
    assert (req1 || req2)
else $error(“Assert for REQUEST state failed”);
```

Такой формат оператора **assert** является наиболее простым. Он совпадает с аналогичным оператором в языке VHDL. Однако далеко не всегда имеется возможность проверить функции устройства, основываясь только на выполнении логических выражений. Возможности System Verilog для верификации существенно расширены именно в части описания последовательностей состояний. На рис. 3 показан пример последовательности сигналов, в которой тактовый сигнал clock вызывает появление сигналов a, b, c.

Для рис. 3 можно записать выражение на System Verilog, описывающее показанную последовательность:

```
a ##1 b ##1 c ##1
```

Символ ## используется для указания задержек в тактах (а не в единицах времени, которые задаются одинарным символом #). Таким образом, приведенное выражение описывает последовательность, в которой появляется сигнал a, на следующем такте должен появиться сигнал b, и еще через такт — сигнал c.

В сложных системах возможно появление сигналов не через строго определенные интервалы времени. Например, периферийное устройство может ответить на запрос через 1 такт, но спецификация шины может допускать ответ и через пять тактов (для медленной периферии). Такое поведение описывается выражением:

```
req ##[1:5] ack
```

В данном случае req и ack — примеры сигналов, а не элементы конструкции. Выражение определяет, что появление сигнала ack возможно в интервале от 1 до 5 тактов после появления req.

Подобные выражения (последовательности) описываются внутри конструкции, определяемой ключевым словом **sequence**:

```
sequence test_seq;
  a ##1 b;
endsequence
```

Вместо максимального значения можно использовать символ $ (то есть req ##[1:$]). В данном случае максимальный интервал в тактах не задается, но подразумевается, что когда-нибудь этот сигнал все же появится.

System Verilog вводит дополнительные синтаксические конструкции для упрощения описания сложных последовательностей. Например, a[*3] эквивалентно выражению a ##1 a ##1 a ##1 (задается три последовательных повторения сигнала *a*). Выражение (a ##1 b) [*1:5] означает, что последовательность a ##1 b должна повторяться от 1 до 5 раз.

Оператор -> описывает повторения состояния сигнала, между которыми возможны пропуски. Например, следующая последовательность выполнится, если в течение трех тактов сигнал a будет равен 1, причем между состояниями единицы возможно любое количество пропусков (нулей):

```
sequence seq1;
  a[->3];
endsequence
```

Для описания последовательностей используются функции, работающие с событиями (аналогом является event в VHDL):

- **$rose**(<signal> [, clock_event]) — возвращает «1», когда сигнал изменился и стал равен «1» (ср. с rising_edge в VHDL);
- **$fell**(<signal> [, clock_event]) — возвращает «1», когда сигнал изменился и стал равен «0» (ср. с falling_edge в VHDL);
- **$stable**(<signal> [, clock_event]) — возвращает «1», если сигнал не изменялся в течение последнего тактового периода;
- **$changed**(<signal> [, clock_event]) — возвращает «1», если сигнал изменялся в течение последнего тактового периода;
- **$past**(<signal>[, [number_of_ticks] [,[expression] [,[clocking_event]]]]) — возвращает значение сигнала в момент времени **number_of_ticks** до текущего момента, если **expression** принимает истинное значение.

Над последовательностями допустимы логические операции. Оператор **and** (логическое И) возвращает истину, когда выполнились обе последовательности, для которых вычисляется этот оператор. Последовательности начинаются в одно и то же время и не обязаны завершаться на одном и том же такте. В отличие от этого оператор **intersect** требует, чтобы обе последовательности завершались одновременно.

Оператор **or** (логическое ИЛИ) возвращает «истину», когда хотя бы одна из последовательностей находится в состоянии «истины».

Оператор **first_match** возвращает истину только в момент первого выполнения условия, заданного в качестве его аргумента.

Оператор **throughout** (дословно — «сквозь») проверяет выполнение некоторого условия в течение всей последовательности. Например, если проверяется корректность обмена по некоторой шине, то ее спецификация может требовать, чтобы сигналы записи и чтения никогда не были в активном состоянии одновременно в течение всего времени предоставления шины этому устройству (за пределами данного интервала их значения неважны). Это можно описать следующей последовательностью:

```
sequence check_bus_access;
  (req) ##[1:5] gnt ##0 ((read ^ write) throughout(grant[*4]));
endsequence
```

Приведенная запись означает, что после запроса шины (req) сигнал подтверждения доступа (gnt) должен появиться в течение 1–5 тактов. Одновременно с этим (##0) должно выполняться условие: активен только один из сигналов read и write, причем в течение четырех тактов, пока действует сигнал gnt.

Оператор **within** проверяет нахождение одной последовательности внутри другой, то есть вложенная последовательность начинается не ранее и кончается не позднее, чем охватывающая. Например, таким способом можно проверить, что стробирующий сигнал находится «внутри» сигнала данных:

```
sequence check_s1_s2;
  s1 within s2;
endsequence
```

Следующей конструкцией для описания логических условий и правил является свойство (property). Свойства предоставляют возможности для описания более сложных соотношений между сигналами, которые не выражаются статическими проверками. Например, с помощью оператора импликации |-> можно проверить наличие корректной реакции схемы на определенные входные воздействия. Оператор импликации внутри свойства может быть записан следующим образом:

```
property prop1(x, y)
  x |-> y;
endproperty
```

Если появляется сигнал x, свойство проверяет наличие вследствие этого сигнала y. Можно обратить внимание, что в описании свойства присутствуют формальные параметры, следовательно, описанные свойства могут быть многократно применены при верификации с разными наборами фактических параметров.

Специальная конструкция **disable iff** служит для описания условия, когда свойства не должны проверяться. Обычно это условие записывается в виде:

```
@(posedge clk) disable iff(reset)
```

Смыслом этой записи является исключение проверки описанного свойства в то время, пока действует сигнал сброса. В момент сброса многие соотношения между сигналами могут не выполняться, поскольку разработчик мог запланировать для них специальное поведение.

Следующий пример описывает свойство, позволяющее проверять сигналы на минимальную длительность. Проверка не производится, если активен сигнал сброса. В теле свойства записан оператор импликации, который утверждает, что при выполнении функции **$rose(a)** (то есть появление фронта на сигнале a) этот сигнал должен оставаться в состоянии логической единицы не менее **num** тактов. Имя сигнала и количество тактов являются формальными параметрами свойства, что позволяет ниже в тесте записать вызов этого свойства для фактического параметра s и убедиться, что его длительность составляет не менее пяти тактов:

```
property min_width(a, reset, num);
  disable iff(reset)
  $rose(a) |-> a[*num];
endproperty

assert property (min_width, s, reset, 5);
```

Моделирование и верификация формируют обширное поле деятельности для разработчика, поэтому без учета специфики конкретного проекта невозможно описать сколько-нибудь исчерпывающую методологию верификации. Важнее здесь то, что современные средства проектирования расширяют список доступных разработчику инструментов для выражения различных подходов и методик. В качестве отправной точки можно ознакомиться со специальной литературой [1, 2].

Заключение

Добавление языка System Verilog в САПР Vivado отражает общую тенденцию все большего распространения средств проектирования на системном уровне. В данном случае возможности System Verilog концентрируются не столько в сфере описания проектов, сколько в сфере их моделирования и верификации, что позволяет разработчикам сократить количество итераций до получения работоспособного устройства, полностью удовлетворяющего спецификациям. Язык привносит ряд особенностей, характерных для языков программирования, поэтому для его эффективного применения может потребоваться дополнительное время. В целом следует отметить, что новые возможности System Verilog концентрируются прежде всего в области моделирования и верификации, поэтому освоение данного инструмента актуальнее для разработчиков, использующих ПЛИС большого логического объема.

Литература

1. https://www.doulos.com/knowhow/sysverilog/tutorial/
2. Bergeron J., Cerny E., Hunter A., Nightingale A. Verification Methodology Manual for System-Verilog. Springer, 2005. ISBN: 0-387-25538-9.