УДК 004

# SystemVerilog утверждения для верификации и имитационного моделирования

А.А. Ветошкин

ЗАО «МЦСТ», alexey.a.vetoshkin@lab.sun.mcst.ru

***Аннотация* — Рассматриваются особенности верификации на основе SystemVerilog утверждений (SVA). Приведены основные подходы к применению SVA. Использование этого метода и других возможностей языка SystemVerilog показано на примере решения задачи оптимизации схемы вычисления адресов операндов. Рассмотрено применение SVA для сбора информации при имитационном моделировании схемы обращения к памяти тегов.**

***Ключевые слова* — SystemVerilog утверждение, SVA.**

## I. Введение

В настоящее время сложность аппаратуры достигла такого уровня, что проверка правильности RTL-описаний немыслима без применения новых методов верификации. Одним из таких методов является верификация, основанная на утверждениях. Этот метод подразумевает расстановку внутри моделируемого кода "закладок", проверяющих работу тех или иных функций по мере их срабатывания, не дожидаясь распространения следствия данного срабатывания на выход схемы. Полная управляемость и наблюдаемость внутренних цепей проекта, обеспечиваемая верификацией с помощью утверждений, значительно, сокращает время отладки. Процент ошибок, который удается обнаружить, используя утверждения, в наибольшей степени зависит от умения выделять в RTL-описании критические участки для проверки с помощью утверждений. Количество написанных утверждений тоже немаловажно. По результатам исследований, проведенных фирмой Cadence [6], использование утверждений позволило выявить в реальных проектах до 90% ошибок.

Основными языками описания утверждений являются OpenVera Assertions (OVA), Property Specification Language (PSL) и SystemVerilog Assertions (SVA). В данной статье описываются подходы к написанию утверждений с использованием языка SystemVerilog, к которому в последнее время наблюдается растущий интерес. Использование возможностей SystemVerilog позволяет повысить эффективность процесса разработки и верификации, что особенно ощутимо при работе со сложными проектами большого объема. Применение этих возможностей показано на примере решения реальных задач верификации и имитационного моделирования. Особое внимание уделено использованию SystemVerilog утверждений.

## II. SystemVerilog утверждения

**Утверждение** – это сжатое описание желательного или нежелательного поведения устройства. Рассмотрим в качестве примера взаимодействие двух устройств (см. рис. 1).

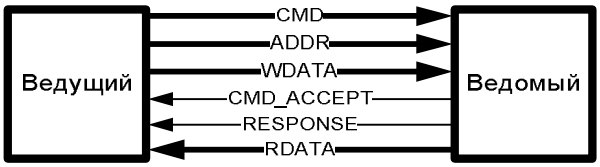


**Рис. 1. Упрощенная структурная схема взаимодействия двух устройств**

Взаимодействие между устройствами начинает ведущий, когда выставляет команду ***CMD*** и адрес ***ADDR***. Если это команда записи, то вместе с командой и адресом ведущий выставляет данные для записи ***WDATA***. Команда удерживается на шине до тех пор, пока ведомый не выставит сигнал ***CMD_ACCEPT***, подтверждая то, что он принял команду.

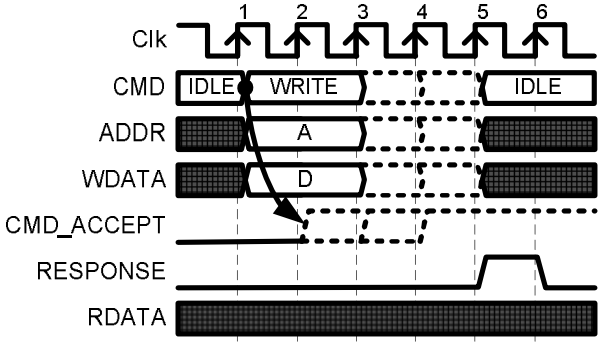


**Рис. 2. Диаграмма, описывающая желательное поведение устройства: сигнал CMD_ACCEPT должен прийти в течение трех тактов после появления команды на шине CMD**

Если транзакция прошла успешно, то ведомый выставляет сигнал ***RESPONSE***. При обработке команды чтения, вместе с сигналом ***RESPONSE***, выставляются данные для чтения ***RDATA***. Для правильной работы необходимо, чтобы ведомый выставил сигнал ***CMD_ACCEPT*** в течение трех тактов после того, как ведущий выставил команду на шине ***CMD*** (см. рис. 2):

```
cmd_accept_wait:
 assert property ( @(posedge Clk)
  (CMD != `IDLE) |-> ##[1:3] CMD_ACCEPT );
```

Язык SVA представляет некое подмножество языка SystemVerilog, что дает возможность применять метод верификации на основе SVA без полного изучения SystemVerilog. Семантика языка SVA схожа с PSL, но есть и ряд новых, по сравнению с языком PSL, возможностей.

Структура языка SVA нагляднее всего представляется, как показано на рис. 3. В основе любого SystemVerilog утверждения лежат *булевы выражения*, которые принимают значения ИСТИНА или ЛОЖЬ. В приведенном выше примере SystemVerilog утверждения используется два булевых выражения: «(CMD != `IDLE)» и «CMD_ACCEPT».


**Рис. 3. Структура языка SVA**

*Последовательности* описывают поведение проекта во времени и строятся на основе булевых выражений. Выражение «(CMD != `IDLE)» – это простая последовательность, состоящая из одного выражения. Сложные последовательности строятся с использованием временных задержек, которые задаются с помощью оператора "##". Построение сложных последовательностей продемонстрировано на примере взаимодействия сигналов "a" и "b" (см. рис. 4).

*Свойства* определяют отношение между логическими выражениями и последовательностями. Для сигналов ***ADDR*** и ***CMD***, изображенных на рис. 2, выполняется свойство: адрес ***ADDR*** не меняется, пока на шине ***CMD*** стоит команда.

```
property stable_addr;
 @(posedge Clk) disable iff (~reset)
 ((CMD != `IDLE) && !CMD_ACCEPT) |=> $stable(ADDR);
endproperty
```

Свойство с именем stable_addr осуществляет проверку по переднему фронту тактового сигнала («@(**posedge** Clk)»). При активном сигнале reset проверка не выполняется («**disable iff** (~reset)»). Проверка того, что адрес не меняется, должна происходить только по событию. В данном случае событием является наличие команды на шине ***CMD***. Для этого используется операция импликации.

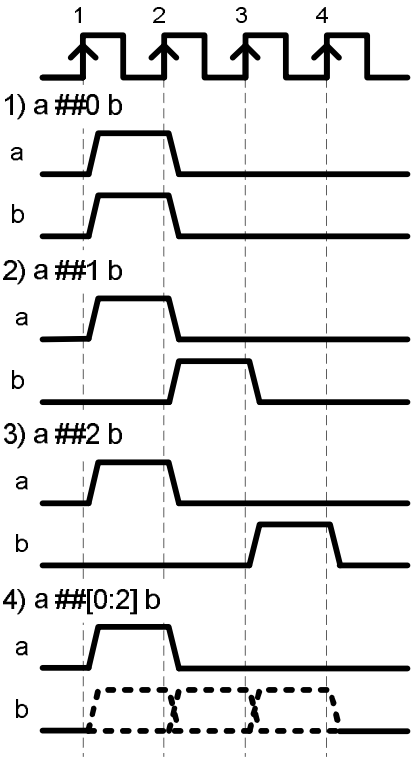


**Рис. 4. 1) Сигнал "b" установится в '1' в том же такте, что и сигнал "a"; 2) Сигнал "b" установится в '1' на следующем такте после "a"; 3) Сигнал "b" установится в '1' через 2 такта после "a"; 4) Сигнал "b" установится в '1' в промежутке 0-2 тактов после "a" (данной последовательности удовлетворяет любая из последовательностей 1)-3))**

Пока к свойству не применена *директива* (assert, assume, cover, expect), проверка выполнения свойства не начинается.

## III. Подходы к применению SVA

Разработка утверждений, которые бы в полной мере проверяли поведение устройства, задача нетривиальная. Одно дело, когда пишутся утверждения для таких распространенных протоколов, как PCI, AMBA, OCP. Зачастую, спецификации к таким протоколам уже содержат описания проверок, которые необходимо реализовать с помощью утверждений, и в дальнейшем требуется только реализация. Если же встала задача разработки утверждений для специфичного устройства, то здесь рекомендуется следующий подход:

Таблица 1

**План верификации**

| **Функциональность для проверки** | **Назначен для написания** |
|---|---|
| Ведомый должен выставить сигнал ***CMD_ACCEPT*** в течение трех тактов после того, как ведущий выставил команду на шине ***CMD*** | Команде разработчиков |
| Адрес и данные не должны меняться, пока не появится сигнал ***CMD_ACCEPT***. | Команде разработчиков |
| Адрес не должен меняться, пока на шине ***CMD*** стоит команда | Команде верификаторов |

- Выделить в своем проекте типовые элементы (автоматы, АЛУ, арбитры, FIFO, памяти). Методика проверки этих блоков с помощью утверждений уже описана [1]-[3]. Остается только применить эту методику с учетом особенностей вашего устройства, тем самым, сэкономив время. Реальные проекты редко состоят из одних типовых элементов, и работы по написанию утверждений все равно будет достаточно.
- Проверка всех комбинаций сигналов невозможна и не нужна, поэтому необходимо выделить в своем проекте ключевые функции, операции, состояния для проверки. Как писалось выше, от этого напрямую зависит эффективность верификации на основе утверждений.
- Выбрать место объявления утверждений. SVA могут быть объявлены напрямую в RTL-описании или вынесены в отдельный модуль [4].
- При тестировании аппаратуры часто встречаются типовые ситуации. Их описание с помощью утверждений можно ускорить, используя специальные библиотеки, которые содержат утверждения, наиболее часто используемые разработчиками. Можно выделить следующие библиотеки: Open Verification Library (OVL) фирмы Accelera, Incisive Assertion Library (IAL) фирмы Cadence, Questa Verification Library (QVL) фирмы MentorGraphics и SystemVerilog Assertions Checker Library фирмы Synopsys. Все эти библиотеки имеют в своем составе схожие наборы утверждений.

Для достижения максимального эффекта рекомендуется использовать библиотечные утверждения настолько полно, насколько это возможно, т.к. эти утверждения уже полностью отлажены и описаны в документации.

- На этапе разработки спецификации следует составить план верификации, который содержит список ситуаций для проверки с помощью утверждений. В идеале, в написании утверждений должны принимать участие и команда разработчиков, и команда верификаторов. Это позволяет выявить некоторые ошибки в логике работы устройства. Утверждения, написанные разработчиком, описывают его предположения о работе устройства. Эти предположения могут быть не всегда достоверными, поэтому желательно вовлечь в процесс написания утверждений верификатора. Описать поведение внутренних сигналов может только разработчик, а верификатор проверяет соответствие устройства спецификации. План верификации для схемы, изображенной на рис. 1, представлен в таблице 1.

Желательно не пренебрегать написанием плана верификации. Это придает работе наглядность, позволяет найти идентичные утверждения. Все ситуации для проверки невозможно выделить на этапе разработки спецификации, также как и определить, кто будет писать то или иное утверждение, поэтому работа продолжается на этапах создания RTL-модели и верификации. Создание плана занимает некоторое время, но является важным этапом для успешного использования SystemVerilog утверждений.

## IV. Оптимизация схемы вычисления адресов операндов

В настоящее время задача создания RTL-описания с нуля встает перед разработчиками гораздо реже, чем модификация уже имеющегося кода. Очень часто в новом проекте приходится дорабатывать и оптимизировать RTL-описание из предыдущего проекта. И тут главным правилом, которым должны руководствоваться разработчики является, “не навредить”. Соблюсти это правило не так-то просто, т.к. любая модификация исходного RTL-описания может повлечь за собой возникновение ошибок. Нередко приходится вносить изменения, при которых логика работы устройства остается неизменной. В основном это изменения, направленные на улучшение результатов синтеза: перевод структурного или потокового описания в поведенческое, изменение разбиения RTL-описания на модули, перемещение комбинационной логики между стадиями конвейера.

Языки описания аппаратуры дают возможность описать одну и ту же схему разными способами. Даже опытный разработчик не всегда может сходу написать код, который при синтезе даст самую быструю схему. При увеличении тактовой частоты от проекта к проекту из RTL-описания “выжимается” все до последней наносекунды. Здесь приходится перебирать различные

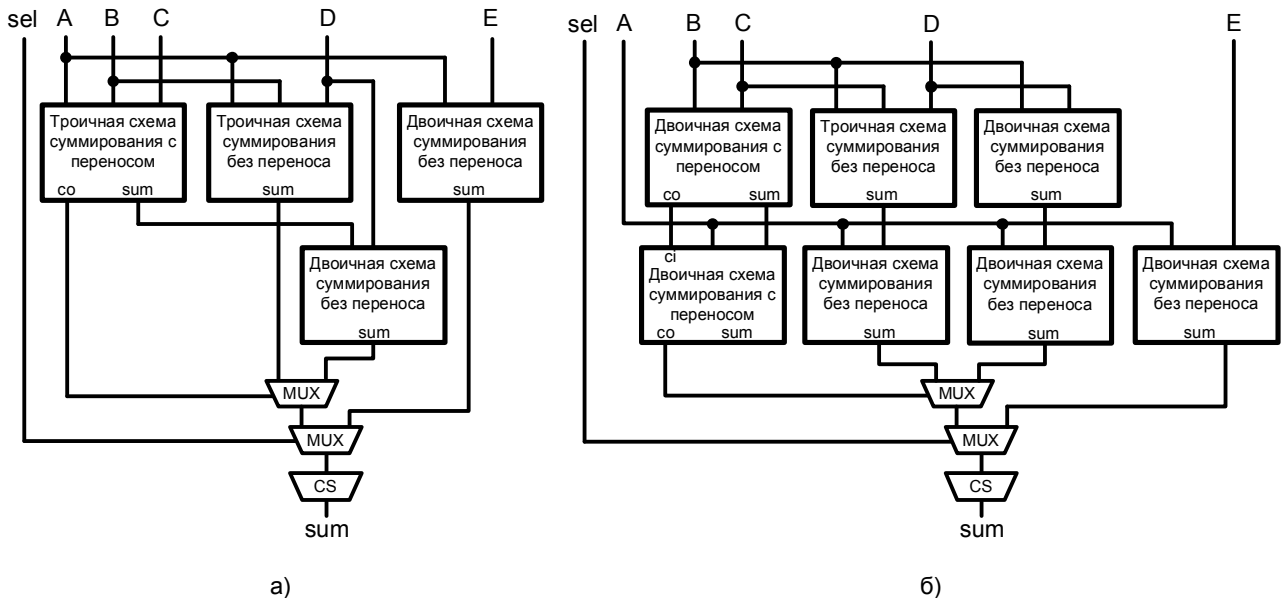


**Рис. 5. а) Схема вычисления адресов операндов; б) Реструктурированная схема вычисления адресов операндов**

варианты, использовать специальные оптимизированные по времени или по площади IP-блоки (например, Design Ware от Synopsys).

На рис. 5а приведена схема вычисления адресов операндов. Согласно ограничениям, накладываемым конвейерной архитектурой, эта схема должна успевать вычислять адрес за время одного такта - 1нс. После того, как было принято решение брать переменную A из памяти, которая вносит задержку в 0.88 нс, схема перестала удовлетворять временным ограничениям. В связи с этим появилась необходимость оптимизировать исходное RTL-описание так, чтобы поведение схемы не изменилось. Оптимизация проводилась в несколько этапов:

1) RTL-описание было переведено из потокового типа в поведенческий.
2) Вместо обычных суммирований в описание были вставлены оптимизированные по времени сумматоры из библиотеки DesignWare. Обычно элементы этой библиотеки используются при синтезе автоматически, однако, для достижения лучшего результата иногда необходимо явно указывать это в описании.
3) Схема была реструктурирована. Все суммирования, которые можно выполнить без берущегося из памяти значения “A”, были вынесены в отдельный модуль (см. рис. 5б).
4) Часть комбинационной логики была перенесена на следующую стадию конвейера.

Синтез проводился с использованием САПР DesignCompiler фирмы Synopsys. Результаты синтеза на каждом этапе представлены в таблице 2.

Убедиться в том, что при внесении изменений поведение схемы не изменилось, можно только перебрав все входные наборы и сравнив состояние выходов старой и новой схемы. Для этого было создано тестовое окружение на языке SystemVerilog (см. рис. 6), которое использовалось на каждом этапе оптимизации.

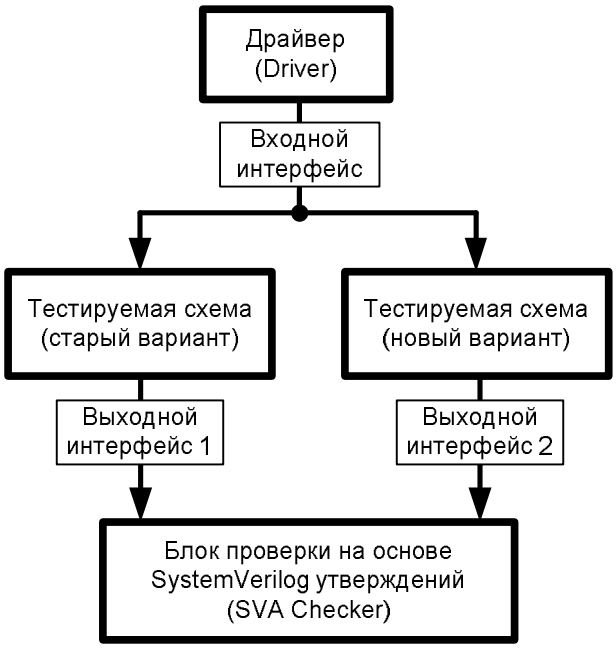


**Рис. 6. Архитектура тестового окружения**

За счет встроенных в SystemVerilog средств объектно-ориентированного программирования (ООП), созданное тестовое окружение обладает возможностью многократного использования. Мощный механизм генерации случайных последовательностей и средства оценки функционального покрытия позволили добиться высокого качества отладки.

Таблица 2

**Результаты синтеза схемы вычисления адресов операндов**

| | **Исходный вариант** | **Этап 1** | **Этап 2** | **Этап 3** | **Этап 4** |
|---|---|---|---|---|---|
| Длина критического пути (Critical Path Length) | 1.38 нс | 1.30 нс | 1.27 нс | 1.05 нс | 0.93 нс |
| Превышение по критическому пути (Critical Path Slack) | -0.44 нс | -0.36 нс | -0.34 нс | -0.1 нс | - |
| Комбинационная логика (Combinational Area) | 1833 эквивалентных вентилей | 1268 эквивалентных вентилей | 1437 эквивалентных вентилей | 615 эквивалентных вентилей | 1142 эквивалентных вентилей |
| Некомбинационная логика (Noncombinational Area) | 3452 эквивалентных вентилей | 3485 эквивалентных вентилей | 3488 эквивалентных вентилей | 3463 эквивалентных вентилей | 2391 эквивалентных вентилей |
| Область ячеек (Cell Area) | 5286 эквивалентных вентилей | 4754 эквивалентных вентилей | 4925 эквивалентных вентилей | 4078 эквивалентных вентилей | 3533 эквивалентных вентилей |
| Выигрыш по времени по сравнению с предыдущим вариантом | - | 0.08 нс (5.8 %) | 0.03 нс (2.3 %) | 0.22 нс (17.3 %) | 0.12 нс (11.4 %) |
| **Суммарный выигрыш по времени** | **0.45 нс (32.6 %)** | | | | |

Идентичность выходов схем на протяжении всего времени симуляции проверялась с помощью блока проверки на основе SystemVerilog утверждений (SVA Checker). В основе блока лежит параметризированное свойство p_equal:

```
property p_equal(old_sig, new_sig);
  @(posedge Clk)
    (old_sig == new_sig);
endproperty
```

За счет того, что сигналы с выходов старой и новой схем передаются в качестве параметров, это свойство можно применять к любому количеству пар выходных сигналов. Для каждой пары необходимо применить директиву assert:

```
ap_sum: assert property (p_equal(old_sum, new_sum));
```

## V. Модель для определения необходимого количества портов на чтение памяти тэгов

В поисках оптимального архитектурного решения разработчикам нередко приходится перебирать множество вариантов. Очень важно определиться с выбором на ранних стадиях проектирования, т.к. в дальнейшем это может значительно сэкономить время. Если решение не может быть получено аналитическим способом, то целесообразно использовать модель, имитирующую ту или иную реализацию.

На рис. 7 представлено два варианта схем обращения к памяти тэгов: 1-портовый и 2-портовый. Обращение к памяти происходит по трем каналам: по основному каналу (main) и по 2-м каналам подготовки переходов (req1 и req2).

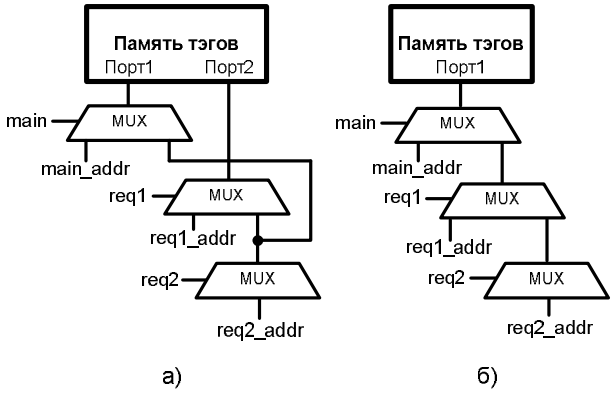


**Рис. 7. а) Схема обращения к 2-портовой памяти тэгов; б) Схема обращения к 1-портовой памяти тэгов**

В ходе исследования необходимо было для обоих вариантов реализации определить, как часто блокируются запросы req1 и req2 при различной интенсивности обращений к памяти. Использование 1-портового варианта выгоднее по ресурсам и проще в реализации, однако, не ясно как это повлияет на производительность. Созданная имитационная модель показана на рис. 8.

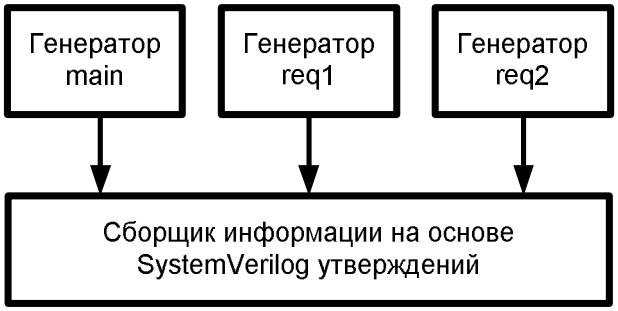


**Рис. 8. Имитационная модель**

Для описания модели необходимы средства генерации потока запросов по различным законам распределения и средства сбора информации с последующей возможностью отображения этой информации в форме таблицы или графика. Наряду с языками имитационного моделирования (GPSS, Simula, SimScript и т.д.), этими средствами обладает SystemVerilog.

На генерацию запросов накладывались специальные ограничения. Для генератора main ограничение выглядит так:

```
constraint main_dist {
  main dist {16:=16, 13:=34, [10:11]:/ 34, 8:=16};
}
```

Это ограничение позволяет генерировать запросы main по нормальному закону распределения. В предельном случае 1 запрос в 8 тактов генерируется с вероятностью 16%, 1 запрос в 10-11 тактов генерируется с вероятностью в 34% и т.д.

В рассматриваемых схемах возникает 3 случая блокировки запроса:

- Блокировка req2 (2-портовый вариант): (req2 && main && req1);
- Блокировка req1 (1-портовый вариант): (req1 && main);
- Блокировка req2 (1-портовый вариант): (req2 && (main || req1)).

Для сбора информации о количестве посланных запросов и возникших блокировок использовались SystemVerilog утверждения с директивой cover:

```
property singleport_req1_block_p;
   @(posedge Clk) disable iff (!active)
         req1 && main;
endproperty

singleport_req1_block:
   cover property (singleport_req1_block_p);
```

Директива cover фиксирует информацию о том, сколько раз выполнилось свойство в процессе симуляции. С помощью специальных средств анализа эта информация считывается из базы данных, и генерируется итоговый отчет, содержащий информацию о количестве выполнений каждого свойства.

Процесс моделирования проводился в течение 500000 тактов. Результаты моделирования при различной интенсивности запросов req1 представлены на рис. 10. Количество блокировок запроса req2 при 2-портовом варианте не превышает 0.5% от общего количества запросов, в то время как при 1-портовом варианте в среднем блокируется 5% запросов. Полученные результаты показывают, что использование 1-портового варианта может негативно сказаться на производительности.

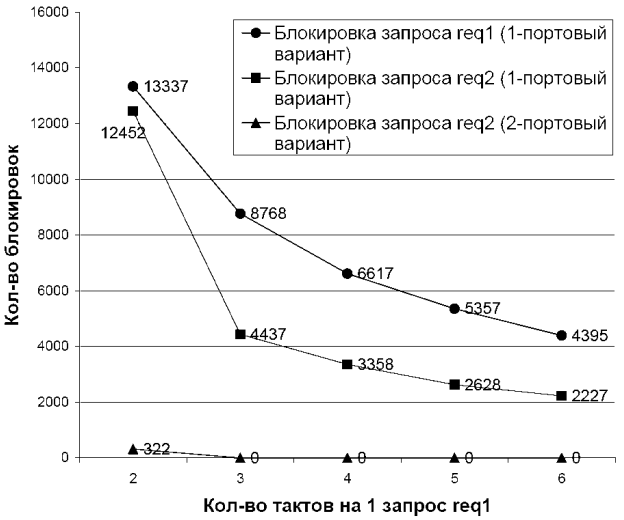


**Рис. 9. Результаты моделирование при различной интенсивности запросов req1**

## VI. Заключение

Применение SystemVerilog утверждений позволяет быстро и эффективно решать различные задачи разработки и верификации цифровых схем. Несмотря на значительные преимущества метода верификации на основе утверждений, разработчики неохотно применяют его при верификации, ссылаясь на то, что утверждения замедляют симуляцию, требуют затрат времени на написание, усложняют RTL-код. С развитием средств САПР и языка SVA влияние вышеуказанных проблем стало незначительным.

На сегодняшний день написание утверждений становится неотъемлемой частью работы над RTL, также как проверка правильности RTL-описаний с точки зрения стандартов кодирования (HDL анализ) или оценка функционального покрытия проекта тестами (coverage анализ).

## Литература


[1] Vijayaraghavan S., Ramanathan M. A Practical Guide for SystemVerilog Assertions // Springer, 2005.

[2] Foster H., Krolnik A. Creating Assertion-Based IP // Springer, 2005.

[3] SystemVerilog Assertions Handbook // VhdlCohen Publishing, 2005.

[4] Clifford E. SystemVerilog Assertions. Design Tricks and SVA bind files // Sunburst Design, inc., 2009.

[5] Долинский М. Assertion Based Verification— верификация, основанная на утверждениях // Компоненты и технологии. 2004. № 9.

[6] Berman V., Marschner E., Schirrmeister F. Assertion based verification: in the context of SystemVerilog. // http://www.cadence.com/.

[7] Материалы сайта: http://www.testbench.in/.