SystemVerilog и улучшение отладки проектов

Иосиф КАРШЕНБОЙМ
iosifk@narod.ru

Несколько слов о «Кратком курсе HDL-кодирования»

«Краткий курс HDL-кодирования», опубликованный в журнале «Компоненты и технологии» (начиная с № 3'2008 и заканчивая № 4'2009), был встречен читателями хорошо. Вот одно из писем читателей: «Очень благодарен Вам за создание такого труда. Нашел для себя массу полезной информации. Спустя пару недель стал более-менее ориентироваться в текстах Verilog, написал небольшой проект (прием данных с использованием автомата состояний), успешно отладил. Я давно хотел освоить «ПЛИСоводство», но никак не мог преодолеть барьер по уровню знаний. Вся литература, которую я видел, представляла собой справочник команд или перевод «хелпа». После прочтения Ваших уроков наконец-то стала вырисовываться целостная картина. По поводу пожеланий, буду рад почитать еще о каких-нибудь тонкостях, стилях программирования, распространенных ошибках и т. д.». Конечно, кроме «Краткого курса HDL-кодирования», есть и другие учебники, но все же приятно чувствовать, что работа сделана не напрасно.

Также было высказано довольно много пожеланий о том, чтобы продолжить «Краткий курс HDL» и рассказать в нем о новом стандарте языка — Verilog-2005, или SystemVerilog [1]. Автор начал готовить материалы по описанию SystemVerilog, но вот что выяснилось в процессе работы. Сейчас есть ряд программных инструментов, поддерживающих стандарт SystemVerilog. К таким продуктам, в первую очередь, относятся те программные инструменты, которые ориентированы на создание больших проектов. Например, таких проектов, как системы на кристалле. И, вместе с тем, традиционные программные инструменты, поставляемые изготовителями FPGA, либо вообще не поддерживают новый стандарт, либо поддерживают его частично. Известны также случаи, когда программные инструменты довольно известной фирмы делают ошибки при обработке корректно написанных файлов. Ситуация эта вполне закономерная. Идет рост уровня программных инструментов. То, что вчера могло считаться «профессиональным», завтра станет «стандартным». Ошибки, без сомнения, будут исправлены. (Кто же не ошибается, когда темп работ очень высок, и конкуренты со всех сторон «давят». Но можно предположить, что появятся новые ошибки, увы…)

Что касается программных инструментов, поставляемых изготовителями FPGA, то тут нужно понимать, что они позиционируют свои инструменты именно для средних проектов, так как их основная задача — поставить продукты как можно большему числу покупателей, а следовательно, цена на эти программные продукты не может быть высока. Но идет плавное повышение уровня. И со временем в программных продуктах изготовителей FPGA, несомненно, появится и поддержка стандарта языка SystemVerilog. Так что же делать? Рано писать о SystemVerilog или уже пора? Ответить на это могут только читатели.

Но, прежде чем начинать углубляться в «дебри», давайте для начала рассмотрим, что же дает применение нового языка именно при отладке проектов. Поскольку мы уже знаем, что именно отладка проектов — это самая трудоемкая операция. И поскольку именно плохая отладка проектов приводит к самым большим потерям при разработке проектов. Таким образом, теперь, после прочтения данной статьи, читатели уже сами смогут решить, насколько их заинтересует новый цикл.

Введение

Большинство фирм, представляющих свои программные инструменты, начинают презентации нового продукта в совершенно одинаковом стиле. Сначала они показывают, что первым инструментом человека были камни и палки. Потом идет переход к паровой машине, и уже следом за ней компании представляют свои компьютерные программные инструменты. Нет смысла развивать подобное введение уточняющими описаниями. Будем считать, что мы довольно быстро прошли эту часть введения. Давайте более внимательно проследим историю развития программных инструментов, применяемых для проектирования FPGA. Эта история тесно связана с ростом технологий изготовления кристаллов. На первом этапе развития шла борьба за увеличение числа транзисторов. На следующих этапах транзисторов в кристаллах стало значительно больше, настолько больше, что основной проблемой теперь стала следующая: «А что мы сможем сделать на кристалле с таким большим числом транзисторов?» Кстати, для тех, кто интересуется такими историческими закономерностями, автор может порекомендовать книгу [2], наглядно показывающую взаимное влияние технологии и человеческих отношений.

Теперь вернемся к технологии описания ресурсов микросхем. На первом этапе это были простейшие средства, представляющие собой таблицы истинности. Следующим этапом в развитии программных инструментов стали языки описания и схемный ввод проекта. Различные языки описания через некоторое время разделились на две группы. В раннюю группу вошли те языки, которыми сейчас занимаются «компьютерные археологи». Из этой группы языков сейчас наиболее известны Абель и AHDL. Конечно, в этом месте ряд читателей начнет высказываться, что вот, мол, AHDL вполне живой. Да, и латынь еще используется врачами… Никто и не спорит. Автор и сам довольно долго и успешно применял AHDL. Но, увы, по поводу AHDL можно смело сказать, что сегодня он годится только для самых малых проектов, не требующих больших процедур для отладки. А затем, с ростом проектов, все же придется переходить на современные языки. О них мы поговорим далее, а сейчас остановимся на схемном описании проектов.

Схемное описание проектов было большим шагом вперед в деле развития технологий проектирования. Схемное описание проекта, или «картинки», — это наиболее простая и доступная технология для небольших проектов. В ней все, несомненно, наглядно и просто. Можно лишь добавить, что при использовании библиотек компонентов, поставляемых фирмой-изготовителем, удается применять параметризированные компоненты. И на этом все преимущества схемного описания проекта заканчиваются. Вместе с тем именно рост числа транзисторов на кристалле привел к тому, что один инженер или даже группа инженеров, работающая в одной фирме, оказались не конкурентоспособными в том случае, если они делали проект сами от начала до конца. Именно рост числа транзисторов и удешевление кристалла привели к тому, что появился рынок «готовых к употреблению» модулей, а именно IP-ядер. Применяя IP-ядра, разработчики выполняют проекты быстрее и качественнее. Сами же по себе IP-ядра выполняются так, чтобы быть наименее платформенно-зависимыми. И в этом случае схемное описание проекта абсолютно не годится, так как оно связано не только с софтовым инструментом, примененным при описании проекта, но даже более того — схемное описание проекта может быть зависимым от версии этого самого программного инструмента. Затем

появляются проблемы перехода к новым версиям программного инструмента, к миграции на другие платформы, к другим симуляторам… И так далее…

Итак, возвращаемся к современным языкам описания. Это, конечно, в первую очередь, VHDL и Verilog. Чем же отличается применение языков описания? В чем их основное отличие от того, о чем мы писали выше? Основные отличия выявляются при разработке и отладке больших проектов. Небольшие проекты отлаживаются по симулятору путем «просмотра зубчиков» на временной диаграмме. И если для этого требуется пусть даже несколько часов, то можно считать, что цель достигнута. Когда же визуальный просмотр диаграмм становится затруднителен, то в таком случае и проявляются все достоинства языка.

Языки описания проекта имеют синтезируемые и несинтезируемые конструкции. Первые представляют собой то, что используется для описания взаимосвязей ресурсов, размещаемых на кристалле. Именно эта часть проекта будет работать как «аппаратное обеспечение, разработанное пользователем». Несинтезируемые конструкции языка предназначены для того, чтобы задавать дополнительные условия компилятору, симулятору, и они позволяют пользователю управлять процессом отладки. Но, в целом, можно сказать, что в случае применения языков описания рано или поздно разработчик от отладки «зубчиков» на временной диаграмме переходит к отладке данных, проходящих через проект.

Для примера представим, что мы хотим отладить автомат, управляющий приемом-передачей кадров сообщений. Пусть каждый кадр представляет собой сотню-другую байт. И мы хотим отправить запрос, получить подтверждение, затем направить запрос на получение данных, получить данные и уже полученные данные мы хотим сравнить с некоторым шаблоном. И так далее, для разной температуры, для разных питающих напряжений, для разной частоты синхронизации… Только при использовании несинтезируемых конструкций языка мы получаем возможность читать данные из файла шаблона, записывать все ответные действия в файл, производить проверку полученных данных на предмет соответствия требуемым параметрам. Кроме того, параметризация проектов и/или их частей позволяет повторно использовать единожды написанные фрагменты кода.

Какой вывод можно сделать из этой части статьи? Поскольку микросхемы становятся все более и более дешевыми, а их «начинка» все более и более увеличивается, то это способствует увеличению сложности проектов. Как следствие, это неизбежно приводит к требованию применить еще более производительный программный инструмент. На рынке появляются новые средства разработки. И именно их применение способствует ускорению и удешевлению разработок. Или, говоря короче, не применил сегодня новые средства, опоздал с выходом на рынок: завтра можешь оказаться на обочине…

Проблемы, возникающие при проверке проекта, увеличиваются вместе с ростом самого проекта

И, тем не менее, несмотря на введение все новых технологий проверки, проблемы, связанные с проверкой сложных проектов и IP, сохраняются. Чем более сложными становятся проекты и чем больше у них функциональных возможностей, тем больше пропасть между сложностью проекта и моментом окончания отладки такого проекта (рис. 1).

Рис. 1. Увеличение сложности и числа проектов неизменно приводило к увеличению количества ошибок

Исследования показали [3], что в значительной части всех проектов при первом выпуске микросхем имеют место ошибки в кремнии, и главная причина этого — функциональные ошибки.

Эти статистические данные показывают «врожденную» трудность в отладке сегодняшних проектов. Сложные блоки, особенно когда они объединены вместе, имеют большое количество состояний, в которых эти блоки могут находиться. И состояния, в которые попадают блоки, являются трудными для воспроизведения при всех возможных условиях, с которыми будут сталкиваться разработчики при реальном применении изготовленного устройства. Одна из главных задач проверки проекта состоит в том, что необходимо проводить испытания проектов во всех крайних случаях функционирования устройства. К этим крайним случаям относятся предельные температуры, скорости распространения сигналов, пониженные и повышенные напряжения питания и напряжения опорных источников. Только при таких испытаниях можно обнаружить глубоко «зарытые» ошибки проекта. Но для более глубокого анализа проекта

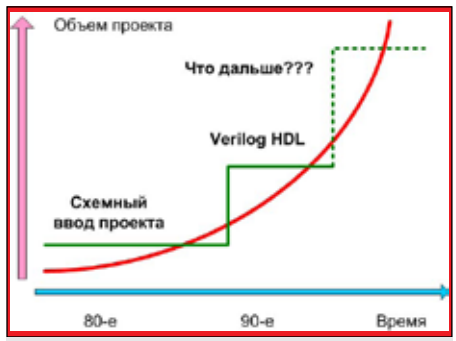

Рис. 2. Применение более совершенных программных инструментов позволяет выполнять более сложные проекты

необходимо или затратить очень много ресурсов, или попытаться применить более новые и более технологичные приемы работы, для того чтобы найти эти ошибки на более ранних этапах проектирования и отладки.

На рис. 2 показана тенденция роста объема проектов (красная кривая) и роста производительности программных инструментов (зеленая линия). Мы видим, что до определенного периода применяемый программный инструмент имеет производительность, позволяющую справляться с проектами. Но с течением времени производительность программного инструмента становится мала. И это вызывает необходимость переходить на более производительный инструмент. Происходит скачкообразный рост производительности программного инструмента. После перехода ситуация на определенный период стабилизируется, а затем неизбежно происходит следующий скачок в производительности программного инструмента.

Традиционные методики проверки

Традиционно тестбенчи основывались на полностью ручной проверке: разработчик сам выполнял все действия, строя тестбенч по методу «прямой проверки». То есть в этих тестах был сделан код, который явно создавал сценарий тестирования, и этот код «прикладывал» стимулирующие воздействия к проекту по определенному адресу или в определенной точке. Например, в тесте указывается конкретный адрес ячейки памяти, которая подлежит проверке в данном тесте. Однако для проверки массива памяти такой способ оказывается малоэффективным. При таком способе проверки все тесты необходимо создавать вручную, поэтому проводят проверку результатов тестирования также вручную или с помощью самопроверок.

«Прямые» испытательные тесты хорошо работают для обычных небольших проектов, но если говорить о больших проектах, например, SoC («системы на кристалле»), то потребуются тысячи испытательных воздействий. Поэтому, если считать (с оптимистической точки зрения), что понадобится до трех дней

на создание и отладку каждого индивидуального теста, то на весь в цикл разработки тестов группа из десяти инженеров-тестировщиков (также оптимистичное предположение) потратила бы год. Единственный способ увеличить производительность проверки состоит в том, чтобы уменьшить время, которое требуется на создание каждого теста.

Новые методики проверки и SystemVerilog

Что же нового можно и нужно применять при разработке тестбенчей? Для повышения эффективности тестирования в новую версию языка Verilog-2005, или SystemVerilog, как он далее будет называться в данной статье, заложены дополнительные возможности для создания сложных проверочных тестов, включая тестбенчи с генерацией случайных воздействий, выполненных с учетом наложенных ограничений, объектно-ориентированное программирование, многопоточные возможности, коммуникацию, синхронизацию и функциональный охват. Эти особенности позволяют пользователям разрабатывать тестбенчи, которые автоматизируют генерацию различных сценариев для проверки.

Автоматическая генерация тестбенчей при рандомизации

SystemVerilog позволяет использовать новые возможности языка для построения автоматизированных тестбенчей. При использовании правильных стратегий по конфигурации среды проверки можно получить значительное преимущество при автоматизации тестирования. И при этом время, требуемое для создания новых тестов, может быть значительно уменьшено. Для таких тестов используется автоматическая генерация случайных воздействий, или рандомизация. Рандомизация тестирования является одним из достаточно мощных средств проверки. То есть в приведенном выше случае будет адресоваться не какая-то конкретная ячейка памяти по непосредственно указанному адресу, а будет адресоваться ячейка по произвольно выбираемому адресу. Причем выбор адреса будет производиться случайным образом.

Но полностью случайный выбор воздействий не всегда может быть полезен. Зачастую необходимо произвести некоторые ограничения на рандомизацию, выполненные с учетом наложенных правил или ограничений, определенных пользователем. Далее мы будем называть такой процесс тестирования как «рандомизацию с ограничениями». И язык SystemVerilog позволяет выполнить такое тестирование. Что это дает? Это дает возможность автоматически генерировать случайные воздействия с учетом наложенных ограничений, и, как следствие, автоматизировать создание новых тестов. Суть таких тестов состоит в том, что при обработке тестбенча дополнительные тесты могут быть автоматически получены из относительно небольшого набора основных тестов: разработчик просто изменяет испытательные параметры или добавляет и совершенствует ограничения. Повышение производительности такого теста показано на рис. 3.

В случае применения «прямого» подхода к тестированию время, требуемое для генерации новых тестов, является относительно постоянным, и, таким образом, возможности по проверке функционирования увеличиваются примерно линейно в течение долгого времени. Для проверки с генерацией случайных воздействий, выполненных с учетом наложенных ограничений, есть некоторый объем работ, который нужно выполнить прежде, чем может быть выполнен первый тест. Этот объем работ представляет собой встраивание в среду проверки возможности параметризовать и ограничивать отдельные части теста. Таким образом, за счет изменения параметров могут быть легко получены новые тесты.

Встраивая рандомизацию не только в значения передаваемых или принимаемых данных, но и в сами сценарии тестирования, можно получать дополнительные тесты, и эти тесты уже могут быть созданы не вручную, а программно. При этом намного более вероятно, что такие тесты произведут проверку всех крайних значений параметров, или, как говорится, проверят все «углы». А значит, с помощью этих тестов можно найти больше ошибок в проекте. Как показано в следующем разделе, такие тесты увеличат охват проекта тестированием и тем самым ускорят проведение его полной проверки.

SystemVerilog имеет все языковые конструкции по проверке, необходимые для испытаний по случайным воздействиям, выполненных с учетом наложенных ограничений. Для SystemVerilog уже накоплен достаточно большой набор рекомендаций о том, как установить такую среду проверки и как использовать объектно-ориентированные методики программирования, а также написать компоненты проверки так, чтобы их можно было повторно использовать для полного набора тестов для проекта и даже для многократного использования в следующих проектах.

Проверка, выполняемая по результатам определения показателей охвата

Что еще может поднять эффективность тестирования? В языке SystemVerilog определены функциональные показатели охвата тестирования. И их тоже можно использовать для автоматической генерации тестов. Самые эффективные методики автоматизированного тестирования включают в себя использование функциональных показателей охвата.

Эти показатели необходимы для определения двух критических моментов при проверке. Первое, что необходимо выявить в процессе тестирования, — это те области проекта, которые были еще недостаточно проверены. Выяснение того факта, что некоторые моменты в проекте недостаточно протестированы и проверены, помогает направить усилия по проверке в нужном направлении. Становится возможным либо добавить к тестированию дополнительные «прямые» тесты, либо изменить нужным образом параметры для испытаний по методу рандомизации с ограничениями.

Показатели охвата также работают как индикатор того, что проверка выполнена достаточно полно, и что проект готов к запуску в производство. Охват обеспечивает больше, чем простой ответ в стиле «да – нет». Изменение динамики роста показателей охвата помогает оценивать стадию выполнения проверки и ее тщательность, что, в конце концов, приводит к выводу о том, что тестирование выполнено и проект готов к запуску в производство. Фактически, показатели охвата играют ведущую роль в проведении автоматизированной и управляемой охватом проверки.

Показатели охвата разделены на две основных категории: охват кода и функциональный охват.

Код представляет собой многообразие различных форм записи — это и строки кода, и ветвления, и различные выражения. Но, тем не менее, проверка охвата кода сейчас является типично автоматизированным процессом, результат которого показывает, был ли весь код в данном описании RTL-проекта задействован в течение этого конкретного моделирования (или при выполнении набора тестов)

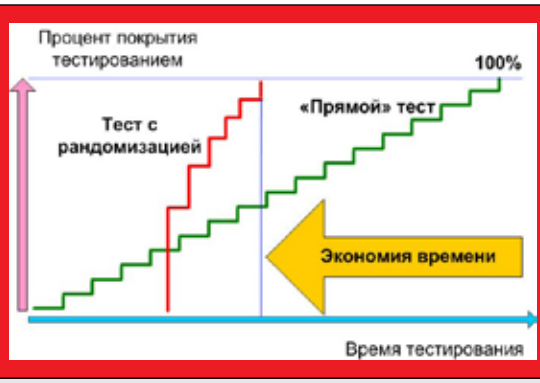

Рис. 3. Автоматизированная проверка намного более эффективна, чем написание «прямых» тестов вручную

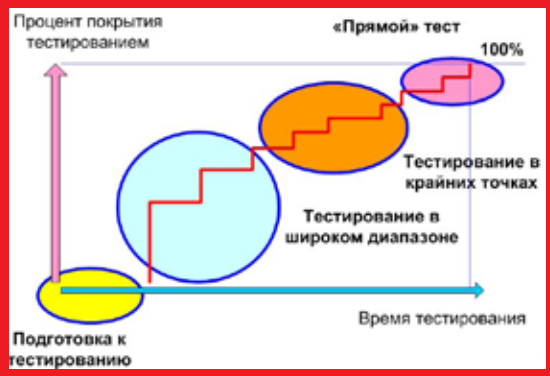

Рис. 4. Автоматизированные и ручные методики применяются в разное время для того, чтобы выявить все «белые пятна» и наиболее быстро закончить процесс проверки

для данного проекта. При этом выявляются «лишние» фрагменты кода, которые не были задействованы при тестировании. Показатели охвата кода — это необходимое, но недостаточное условие для методологии надежной проверки.

В отличие от охвата кода, функциональный охват представляет собой показатель того, сколько проверок было фактически осуществлено из всех запланированных функциональных возможностей проекта. Уверенность в том, что проверки выполняются правильно, может быть еще более увеличена при использовании методик охвата с перекрытием, чтобы измерить комбинации нескольких показателей охвата. Важность функционального охвата и охвата с перекрытием настолько велика, что они должны быть заложены в план проверки проекта еще в начале работы над ним.

Процесс заполнения «белых пятен» при проверке определяется достижением показателей охвата верхней планки заданного диапазона для охвата. Управляемый охватом процесс тестирования и направлен на достижение заданного диапазона охвата. Поэтому, когда будет достигнут 100%-ный охват для всех показателей, можно будет сказать, что проверка произведена достаточно полно. SystemVerilog обеспечивает все возможности для покрытия тестированием всех пунктов охвата от низшего уровня до покрытия группы для высокоуровневых значений и поддержку охвата с перекрытием.

При проверках обычно имеет смысл применять различные подходы для тестирования, которые могут использоваться в различных пунктах в процессе проверки для того, чтобы улучшить охват тестированием. На рис. 4 показан более подробно график процесса проверки по методу рандомизации с ограничениями, взятый из рис. 3.

Первая часть процесса представляет собой разработку основы тестбенча. Это минимально необходимые трудозатраты для проекта, без которых невозможно проводить какие-либо испытания системного уровня. Однако, если проектировщики в это же время параллельно выполняют и описание проекта на RTL-уровне, то они могут для начала выполнять «прямые» тесты уровня блоков или формальный анализ написанного кода.

Как только основа тестбенча готова, группа разработчиков может начать выполнять тесты по методу рандомизации с ограничениями и анализировать первый набор пунктов охвата. В этом пункте тесты имеют обычно весьма широкие диапазоны, генерируя множество откликов в поведении проекта. Когда число «белых пятен» сокращается, то для того чтобы заполнить оставшиеся «белые пятна», приходится приложить уже значительно больше усилий. Инженеры-тестировщики имеют тенденцию сосредотачиваться на специфических пунктах охвата, которые представляют собой пункты охвата для крайних случаев параметров (обычно их называют «углами»). В этом случае приходится тщательно изменять ограничения и параметры, при помощи которых можно генерировать тесты, «покрывающие» эти пункты. «Прямые» тесты также могут играть свою роль в управляемой охватом среде проверки. Хотя метод испытаний с рандомизацией и ограничениями имеет приоритет, но, чтобы заполнить какое-нибудь «белое пятно» в охвате, может быть, проще написать «прямой» тест, чем автоматически генерировать тест, используя методику рандомизации с ограничениями. Цель состоит в том, чтобы у определенных показателей достигнуть 100% охвата любыми соответствующими методами.

Поскольку SystemVerilog представляет собой язык, на котором можно и выполнять проект, и производить проверки, то он является источником всей информации по охвату в течение процесса проверки проекта.

Утверждения

Следующий шаг — это расширение возможности любой из описанных выше методик тестирования при помощи специальных утверждений (Assertions), которые позволяют еще более качественно проверить поведение проекта и сравнить его с намерениями проектировщика, оказывая тем самым существенную помощь в диагностировании ошибок.

Когда-то, в те «далекие» времена (после некоторого размышления, автор поставил здесь кавычки, ибо на самом деле эти времена не так уж и далеки от нас), когда один разработчик полностью владел информацией о поведении проекта, ему не требовалось затрачивать свои усилия на то, чтобы как-то это «зафиксировать». Достаточно было просто взглянуть на временные диаграммы в характерных точках, и разработчик визуально определял, насколько правильно или неправильно ведет себя проект.

Однако появились блоки, используемые как «черные ящики». Причем это могут быть как покупные IP-ядра, так и блоки, выполненные разными инженерами, но из одной группы разработчиков. И в таком случае инженер уже не полностью владеет информацией о том, насколько правильно или неправильно ведет себя заимствованный блок. Чтобы избежать такой ситуации, каждому из разработчиков пришлось затратить дополнительные усилия и поместить сведения о поведении своего тестируемого блока непосредственно в разрабатываемый файл. Таким образом, инженер может и должен заложить в свой проект набор утверждений, которые покажут ожидания разработчика при тестировании проекта. Поэтому, когда разработчик создает свой RTL-код, он должен сразу же закладывать в проект утверждения с соответствующими условиями, которые можно рассматривать как дополнительную документацию по поведению проекта. Эти утверждения покажут то, что разработчики ожидают от поведения проекта при работе данного блока вместе со смежными блоками.

Утверждения могут находиться в проекте на нескольких уровнях иерархии, то есть они могут быть на самом низком уровне и сообщать о том, как должны себя вести определенные модули. Утверждения могут быть и на самом высоком уровне, показывая то, как информация должна проходить через весь проект от начала до конца.

Утверждения могут быть определены разными способами, включая общие RTL-выражения, специальные утверждения HDL-языков и встроенные конструкции утверждений в SystemVerilog. Подход SystemVerilog идеален, потому что утверждения могут быть определены непосредственно в пределах среды проверки или в пределах проекта RTL. Утверждения SystemVerilog позволяют расширить проверку с помощью трех важных способов:

- Они обеспечивают документацию функционирования, которую намерен получить разработчик проекта. Это может быть очень полезно, если проект многократно используется другими разработчиками. Особенно если данный проект помещается в архив проектов для более позднего использования или передается кому-либо по лицензии как коммерческий IP-продукт.

Рис. 5. Утверждения представляют собой основу для проверки

- Симуляторы, поддерживающие конструкции утверждений SystemVerilog, могут в течение моделирования с «прямыми» или случайными тестами выполнить утверждения. Утверждения при моделировании позволяют увидеть прохождение процесса проверки и внутреннего поведения проекта, делая отладку проекта более эффективной.
- Утверждения SystemVerilog можно прочесть с помощью формальных инструментальных средств анализа, которые используют математические методы, чтобы доказать, что каждое утверждение или всегда выполняется, или находить контрпример, показывающий, как утверждение может быть не выполнено. Это позволяет создать простой переход от утверждений в моделировании к более всесторонней проверке и повышает роль формального анализа при запуске проекта в серию.

Утверждения могут влиять на многие части процесса проверки, как показано на рис. 5. В дополнение к моделированию и формальной проверке некоторые формы утверждений могут быть выполнены на аппаратных средствах, эмуляторах — прототипах на основе FPGA, или даже на SoC, которую и разрабатывают в данном проекте, что приводит к значительному росту производительности при моделировании.

Могут быть установлены дополнительные связи между спецификациями протокола на основе утверждений и инструментальными средствами автоматизации тестбенчей, которые поддерживают испытания по методу рандомизации с ограничениями. Утверждения могут также быть задействованы в учете показателей охвата и объединены с другими формами охвата.

В SystemVerilog обеспечен единый механизм спецификации утверждений, который работает со многими инструментальными средствами. Он позволяет утверждениям быть ключевой частью методологии проверки. Широкие возможности языка SystemVerilog позволяют ему соответствовать многим методикам проверки, и все эти методики можно наиболее эффективно использовать вместе. Правильная комбинация расширенных методик проверки может улучшить тщательность проверки, увеличить ее производительность, ускорить процесс разработки при меньшем потреблении ресурсов, чем методы решения «в лоб». Применение языка SystemVerilog может расширить существующие подходы и сформировать базу для всесторонней среды проверки, которая основана на полном преимуществе автоматизации, функционального охвата и утверждений.

«Ложка дегтя»

Обычно, когда какой-нибудь автор пишет, что, мол, все чудесно и безоблачно, искушенный отечественный читатель сразу же начинает сомневаться. Особенно, когда раздаются призывы: «Бери скорее, на всех не хватит!»… Где-то же должен быть подвох! К сожалению, и в деле разработки проектов не обходится без небольшой «ложки дегтя». Фирмы, производители программных инструментов, так же как и разработчики проектов, находятся в состоянии бесконечной гонки. Ибо, если ты не успел выйти на рынок первым… Дело известное. А ошибки неизбежны в любом деле. И именно поэтому так много постов в конференции такого примерно содержания: «Пытался перейти на новую версию компилятора (здесь автор сознательно не называет имя продукта и название фирмы, ибо данное высказывание можно отнести практически к большинству игроков «первой линии»), но, к удивлению, то, что без проблем «компилилось» на предыдущей версии, теперь ни за что не «компилится» в новой версии». Часть таких высказываний связана только с тем, что новые версии программных инструментов зачастую требуют других настроек компилятора. Но другая часть этих высказываний действительно связана с ошибками в программных инструментах. Это подтверждают и фирмы-изготовители. Единственное, что радует в таких постах, так это то, что именно наши соотечественники оперативно находят ошибки и указывают на них изготовителям. То есть в этом случае смело можно сказать, что именно они работают на передовом уровне технологии.

Конечно, применять новые программные инструменты необходимо. Но, вместе с тем, необходимо всегда проверять себя и новый софт на корректность работы. И необходимо учитывать то, что при переходе на новые программные инструменты потребуется выделить определенный ресурс времени на освоение новых инструментов и проверку корректности их работы.

Заключение

Язык SystemVerilog обеспечивает все конструкции и имеет все возможности для того, чтобы построить расширенную среду проверки, охватывающую генерацию случайных воздействий, выполненных с учетом ограничений, проверку, управляемую охватом, и утверждения. Применение языка SystemVerilog может улучшить тестирование, автоматизировать рутинные операции по составлению тестов и увеличить охват тестированием проекта. ■

Литература

1. IEEE Std 1800-2005. IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language.
2. Мак-Нил У. В погоне за мощью. Технология, вооруженная сила и общество в XI–XX веках / Пер. с англ. Т. Ованисяна. М.: ИД «Территория будущего», 2008.
3. Anderson T., Bergeron J., Cerny E., Hunter A., Nightingale A. SystemVerilog reference verification methodology: Introduction. http://www.eetimes.com/showArticle.jhtml?articleID=183702807