УДК 004.052.42

Использование UVM для автономной верификации цифровой аппаратуры

М.М. Ровнягин

ЗАО «МЦСТ», rovnyagin@gmail.com

***Аннотация* — Дается краткий обзор методов автономной верификации. Рассматриваются особенности применения UVM (Universal Verification Methodology) для верификации блоков процессора (СнК) «Эльбрус-2S». Приводятся различные подходы к применению механизмов, реализованных в UVM на конкретном примере. Оцениваются возможности окружений, построенных на базе UVM, к модернизации и повторному использованию.**

***Ключевые слова* — автономная верификация; UVM.**

I. Введение

В настоящее время в связи с увеличением сложности разрабатываемой аппаратуры существенно возросла роль верификации в процессе проектирования цифровых устройств. Для ускорения процесса производства новых устройств применяются современные методы оценки правильности реализации: используются абстрактные модели на языках высокого уровня (ЯВУ), современные методы генерации случайных воздействий для системных тестов, развиваются методы автономной верификации.

Рис. 1. Виды автономной верификации

Выбор метода (см. рис. 1) автономной верификации во многом зависит от сложности тестируемого устройства и сроков, устанавливаемых заказчиком. Если существует потребность в верификации относительно простого модуля в сжатые сроки, возможно применение даже такого устаревшего метода как прямое тестирование (direct testing). Прямое тестирование включает в себя создание набора тестовых векторов для проверки всех интересующих ситуаций, указанных в плане тестирования, написанном по соответствующей спецификации. При тестировании аппаратного блока этим методом получают выходной набор векторов, который сверяют с ожидаемыми значениями. Однако наиболее распространенным способом верификации сейчас является тестирование с использованием тестового окружения. Построение подобной «инфраструктуры» вокруг тестируемого модуля требует определенных затрат по времени, но в итоге, позволяет добиться заметного преимущества по сравнению с методикой прямого тестирования [1]. Тестовое окружение, позволяющее проводить поиск ошибок на основании генерации псевдослучайных тестов, обладает следующими свойствами:

1) **Модульность**. Тестовое окружение представляется набором модулей, выполняющими некоторые законченные функции.

2) **TLM** (Transaction Level Modeling). Все модули работают не с конкретными интерфейсными сигналами, а с транзакциями, которые играют роль высокоуровневых данных. Примерами таких транзакций могут служить процессорные инструкции, сетевые пакеты и т.д.

3) **Повторное использование**. Все модули в тестовом окружении проектируются таким образом, чтобы иметь возможность быть использованными повторно. Для этих целей вводятся разнообразные параметры, влияющие на внутреннюю структуру модуля, но оставляющие определенную функциональность.

4) **Самопроверка**. Эта концепция требует от тестового окружения полностью автоматической проверки. В идеальном случае отработанный тест должен выдать статус о завершении и причину возможного сбоя.

5) **ООП** (Объектно-ориентированное программирование). Тестовое окружение такой сложности может быть написано только с использованием парадигм объектно-ориентированного программирования, что привносит некоторые давно устоявшиеся программные подходы (шаблоны проектирования).

Для написания тестовых окружений можно использовать как высокоуровневые языки программирования, так и непосредственно языки описания аппаратуры.

К первой группе можно отнести инструмент CPP TESK Hardware Edition [2] (созданный в ИСП РАН). Он предназначен для автоматизированной разработки тестовых систем для HDL-моделей аппаратуры. В терминах CPP TESK воздействия, подаваемые на тестируемый модуль, называются стимулами, а результат его работы – реакцией. Используя фреймворк CPP TESK и придерживаясь общей концепции UniTESK, можно максимально ускорить и стандартизировать процесс разработки тестового окружения и модели. Сложная система классов CPP TESK позволяет автоматизировать процесс создания генератора стимулов, эталонной модели и тестового оракула, проверяющего правильность реакций. Подсистема оценки полноты тестирования (называемая «обходчик») обеспечивает проверку переходов между состояниями модели устройства.

Для унификации в области проектирования тестовых окружений с использованием языков описания аппаратуры (SystemVerilog [3]) часто используются специальные методологии верификации. Наиболее популярные из них: UVM (Universal Verification Methodology [4]), OVM (Open Verification Methodology), VMM (Verification Methodology Manual). Каждая из этих методологий предоставляет верификатору определенные правила и инструментарий для разработки тестовых окружений.

II. Компоненты UVM-окружения

Методология UVM была разработана в 2010 году на основе методологии OVM (совместная разработка корпораций Cadence и Mentor Graphics). Она представляет собой набор классов на языке SystemVerilog. При построении окружения используется так называемое моделирование на уровне транзакций, Transaction-Level Modeling (TLM). Смысл этой технологии заключается в абстрагировании от конкретных интерфейсов, которые зачастую являются сложными для работы и понимания. Вместо набора сигналов верификатор оперирует набором более простых пакетов. Пакеты передаются от одного блока к другому через специальные порты. Типичное тестовое окружение изображено на рис. 2.

Основные компоненты тестового окружения UVM (OVM):

1) **DUT** (Device Under Test) – RTL-модель тестируемого устройства.

2) **Пакеты данных** (sequence items) – структуры данных, подаваемые на вход тестируемого устройства. Могут содержать различные сигналы, инструкции, другие пакеты.

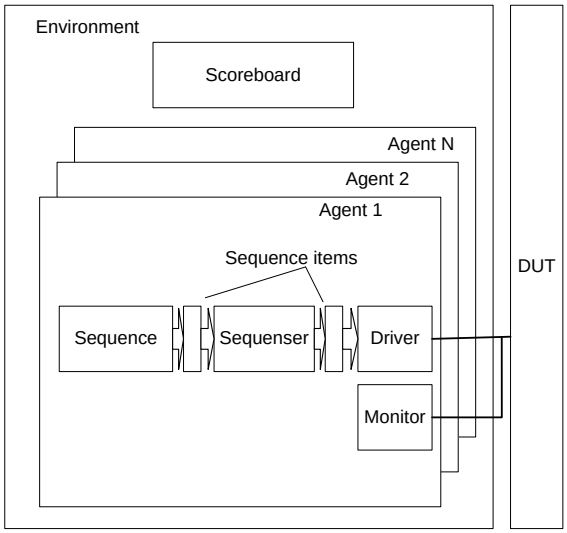

Рис. 2. Состав тестового окружения UVM

3) **Драйвер** (driver) – компонент, непосредственно подающий сигналы в тестируемое устройство. Драйвер получает пакеты данных и обрабатывает их. После этого он выполняет одну или несколько процедур пересылки данных в устройство.

4) **Описатель последовательности** (sequence) – в зависимости от устройства и тестов он может генерировать как произвольные, так и настраиваемые пакеты.

5) **Генератор (сиквенсер)** последовательностей (sequencer) – компонент, контролирующий выполнение последовательностей и передающий пакеты данных в драйвер.

6) **Монитор** (monitor) – компонент, который собирает результаты работы устройства. Как и драйвер, монитор имеет доступ к интерфейсу.

7) **Компаратор** (scoreboard) – компонент, анализирующий полученные результаты работы устройства, проводит оценку правильности функционирования.

8) **Агент** (agent) – компонент, содержащий генератор последовательностей, драйвер и монитор. Агент представляет собой универсальный компонент и используется для уменьшения времени разработки новых окружений, так как разработчику не требуется разбираться в работе его составляющих и подключать их к проекту поодиночке.

9) **Окружение** (environment) – компонент самого высокого уровня. Состоит из набора агентов и других более низкоуровневых компонентов.

III. Подходы к применению UVM

В рамках универсальной методологии верификации существует множество механизмов, обеспечивающих гибкую настройку тестового окружения в зависимости от особенностей устройства. В большинстве случаев, когда устройство «простое» и имеет небольшое число интерфейсов, достаточно создать агенты для каждого из интерфейсов и простейший виртуальный сиквенсер, управляющий работой окружения в целом.

Существуют ситуации, когда такой подход оказывается неэффективным. Например, иногда интерфейсы отдельно взятого устройства могут обладать слабой информативностью, что затрудняет обработку реакции тестируемого модуля на входные воздействия.

В рамках проекта по созданию высокопроизводительного многоядерного процессора «Эльбрус-2S», в частности, выполняется автономная верификация модулей кэша второго уровня микропроцессора с использованием UVM. Одним из таких устройств является очередь отложенных запросов в память (Write Back Queue, WBQ [5]).

Основной задачей WBQ является обработка запросов Write Back (WB), которые генерируются в том случае, когда требуется удалить строку из кеша и записать в оперативную память. Почти всегда причиной возникновения WB являются CPU-запросы на чтение данных, которые при промахе в L2 кэше освобождают место для требуемых данных, вытесняя “старую” строку в память. WB выполняется в 2 этапа. Сначала соответствующий банк L2 кэша генерирует WB-запрос, который через WBQ поступает в арбитр кэша (L2_Arbiter). На следующем этапе арбитр по WB-запросу считывает данные из кэша и отправляет их в память. Для связи с памятью каждое ядро имеет отдельное устройство – Memory Access Unit (MAU).

Помимо обработки WB-запросов, WBQ поддерживает работу с запросами из STMB (STore Miss Buffer) и запросами поддержания когерентности (snoop). Большое количество «соседних» с WBQ устройств (32 интерфейса, 6 устройств), высокая сложность организации взаимодействия между ними, а также малая информативность линий интерфейса между WBQ и L2_Arbiter усложняют проектирование окружения.

Структурная схема тестового окружения приведена на рис. 3. На ней штриховыми стрелками показана основная последовательность обработки STMB и WB-запросов. Такую последовательность можно было бы легко организовать с помощью виртуального сиквенсера, если бы в интерфейс с L2_Arbiter разработчики устройства добавили дополнительные линии, сигнализирующие о типе выдаваемого WBQ запроса (работа с половинной или целой кэш-строкой).

От того, какого типа запрос, зависит последовательность его обработки: L2_Arbiter, принимая STMB запрос на считывание половины кэш строки, выставляет один сигнал об успешной передаче (taken). В процессе приема транзакции на считывание целой кэш-строки сигнал taken выставляется дважды.

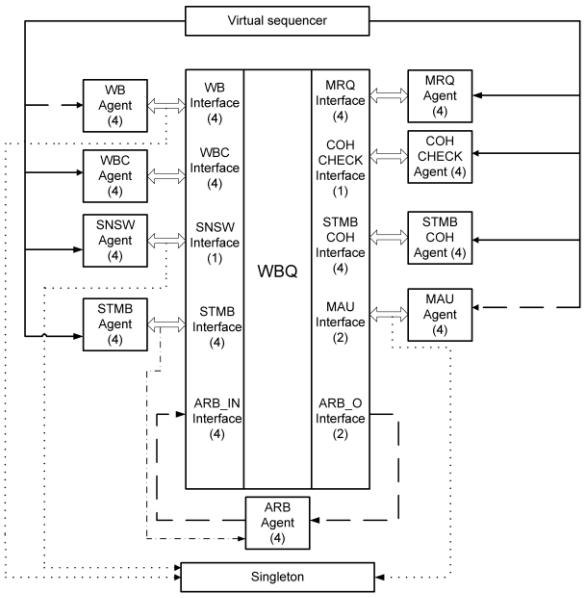

Рис. 3. Тестовое окружение для WBQ

Проблема была решена с помощью использования механизма портов (analysis_port) между драйверами STMB и ARB (показан на рис. 3 штрихпунктирной линией). Передав запрос на чтение половины или целой кэш-строки, драйвер STMB информирует об этом ARB-драйвер. Получив по ARB_O-интерфейсу пакет, ARB-драйвер проверяет список полученных по ovm_analysis_port уведомлений и устанавливает на ARB_O и ARB_IN нужное количество сигналов taken и vdat соответственно.

Применение подобного механизма позволило решить проблему формирования адреса snoop-запроса SNSW-драйвером. В некоторых тестах адрес snoop-запроса должен совпадать с адресами WB-запросов, находящихся в WBQ на момент получения snoop-запроса устройством. Создан класс Singleton, экземпляр которого содержит информацию о всех WB-запросах, находящихся в устройстве WBQ. Перед тем, как SNSW-драйвер устанавливает значение на адресную шину, он выбирает произвольный адрес из массива, хранящегося в Singleton. Обновление информации в Singleton обеспечивают драйвера MAU и WB (связи изображены на рис. 3 пунктирными стрелками).

Использование вышеописанных методов управления процессом тестирования позволяет организовать проверку сложных тестовых случаев. Однако не стоит «злоупотреблять» передачей информации непосредственно от драйвера к драйверу. Практика показывает, что это приводит к сильному усложнению тестового окружения и ухудшает понимание программного кода другими разработчиками. По возможности, предпочтительнее осуществлять передачу информации посредством виртуального сиквенсера.

IV. Автоматизация процесса разработки и поддержки тестового окружения UVM

Жизненный цикл тестового окружения, построенного согласно концепции UVM, включает в себя несколько этапов: создание базовых компонентов тестового окружения, проектирование механизма генерации тестовых воздействий, создание тестов для покрытия тестовых случаев, поддержка тестового окружения, возможное перепроектирование окружения для использования с новой версией аналогичного устройства.

Благодаря модульности UVM-окружения вносить изменения в тестовую инфраструктуру достаточно несложно. Однако при верификации сложных устройств с большим количеством интерфейсов приходится изменять множество файлов-модулей тестового окружения (например, описателей формата пакетов, драйверов, мониторов и т.д.). Автоматизация генерации структуры окружения позволяет существенно ускорить темп разработки верификационных модулей.

Для автоматической генерации компонентов тестового окружения разработана скриптовая программа на языке Perl. Она анализирует переданные через командный интерфейс параметры и имена интерфейсов, после чего по файлам-шаблонам создает необходимые для построения тестового окружения файлы. В результате применения этой утилиты удалось избавиться от большого количества монотонных действий по созданию структуры тестового окружения.

Проблема поддержки тестового окружения решена путем дальнейшего развития идей, заложенных в вышеописанную утилиту и программы, использовавшиеся для генерации тестовых окружений на языке C++ ранее.

Новое приложение анализирует головной файл верифицируемого устройства и предлагает пользователю в графическом режиме распределить входные и выходные сигналы по интерфейсам и задать необходимые типы данных для TLM-пакетов. После чего происходит генерация файлов тестового окружения, в которые инженер-верификатор вносит необходимые правки и описывает протоколы взаимодействия с устройством в файлах мониторов и драйверов. При необходимости создается виртуальная последовательность для управления сиквенсарами интерфейсов.

В процессе тестирования устройства верификатор, как правило, проводит испытания «рабочей копии устройства». Эта копия загружается на рабочую станцию или сервер и периодически обновляется при помощи системы контроля версий (например, SVN [6]). Программа анализирует содержимое top-файла устройства обновленной рабочей копии и, если произошли какие-либо изменения входных или выходных сигналов, предлагает верификатору изменить или откорректировать состав интерфейсов устройства. Если произошло добавление новых линий в состав интерфейса, программа вносит соответствующие дополнения в файлы тестового окружения (файл с описанием интерфейсов, пакетов, последовательностей и т.д.). В случае, если был добавлен новый интерфейс, программа создает необходимые файлы тестового окружения (аналогично случаю генерации). Вносимые изменения по умолчанию выполняются в виде комментариев, которые в дальнейшем «одобряются» верификатором.

Предложенный подход позволяет облегчить задачу разработки и поддержи тестового окружения. В настоящее время наблюдается большой интерес к системам автоматического построения тестовых окружений. Стоит особо отметить, что создание полностью автоматической системы невозможно осуществить, не имея формализированной спецификации устройства. Аналогичные подходы в настоящее время используются в области формальной верификации [7].

V. Заключение

Применение универсальной методологии верификации для автономного тестирования модулей цифровых устройств позволяет: уменьшить время, затрачиваемое на разработку тестового окружения, реализовать генерацию сложных тестовых воздействий для проверки определенных тестовых случаев, обеспечить хорошие показатели повторного использования компонентов тестового окружения.

Автономная верификация, основанная на вышеописанной методологии, в настоящее время является наиболее распространенной и продолжает совершенствоваться (создаются новые системы сбора информации о функциональном покрытии, такие как iTBA и пр.). Комплексное использование UVM, систем сбора информации о покрытии, механизма SystemVerilog Assertions [8] для автономной верификации позволит качественно повысить надежность разрабатываемой цифровой аппаратуры.

Литература

[1] William K. Hardware Design Verification: Simulation and Formal Method-Based Approaches. Prentice Hall PTR, 2005. 624 p.

[2] Интернет-ресурс ИСП РАН. URL: www.unitesk.ru.

[3] Chris S. SystemVerilog for Verification. Springer Science+Business Media, 2008. 400 p.

[4] Mentor Graphics UVM/OVM Documentation (verification methodology cookbook). 2011. 166 p.

[5] Документация на WBQ кэша L2 системы на кристалле «Эльбрус-2S».

[6] Pilato C., Collins-Sussman B., Fitzpatrick B. Version Control with Subversion. O'Reilly Media, 2008. 430 p.

[7] Mahajan Y., Chan C. Verification Driven Formal Architecture and Microarchitecture Modeling // Formal Methods and Models for Codesign, 2007. 5th IEEE/ACM International Conference. P. 123–132.

[8] Clifford E. SystemVerilog Assertions. Design Tricks and SVA bind files. Sunburst Design, 2009. 42 p.