УДК 004.415.538, 004.434

Автоматизация процесса создания тестовых окружений, обеспечивающая сквозной маршрут разработки, верификации и исследования СФ-блоков и СнК

К.А. Жезлов, Я.С. Колбасов, А.О. Козлов, А.В. Николаев, Ф.М. Путря, С.Е. Фролова

ОАО НПЦ «ЭЛВИС», fputrya@elvees.com

***Аннотация* — Основные акценты в статье делаются на автоматизацию процесса создания инфраструктуры для функциональной верификации СФ-блоков. Под инфраструктурой в первую очередь понимаются тестовые окружения, средства сборки, запуска и анализа результатов тестов, а также средства оценки эффективности работы СФ-блоков и систем. Отличительной чертой описываемой в статье инфраструктуры для верификации СФ-блоков является переносимость тестов между этапами верификации и высокая степень повторного использования компонент среды, в том числе таких, как средства сборки тестов, окружений и анализа результатов тестов.**

***Ключевые слова* — СнК, верификация, тестовые окружения, генерация, унификация, параметрический контроль.**

I. Введение

Функциональная верификация, являющаяся критическим этапом маршрута разработки СнК, для современных систем в обязательном порядке дополняется процессами оценки реализуемости целевых задач на проектируемой системе и подтверждения её эффективности для всех целевых режимов работы (в ряде источников данный процесс называют валидацией [1]), что делает этап верификации ещё более сложным и ответственным. В предыдущей работе авторов [2] был обозначен платформенный подход, позволяющий сократить время, затрачиваемое на верификацию. Определено многомерное пространство этапов верификации СФ-блоков и объекты (среди них тестовые последовательности, верификационные компоненты и средства их сборки), упрощение переноса которых между этапами в заданном пространстве является необходимым требованием для ускорения процесса функциональной верификации СФ-блоков и систем.

На конференции МЭС-2014 авторами был представлен ряд работ [3-5], посвящённых созданию элементов инфраструктуры верификации СнК и СФ-блоков. В данных работах была показана необходимость как автономной верификации СФ-блока, так и его верификации в составе системы комплексными и нагрузочными тестами. Особое внимание при этом уделялось унификации и стандартизации верификационных компонент и тестовых сценариев для минимизации затрат на портирование кода между этапами верификации. В данной работе акцент будет сделан на следующем шаге на пути решения проблемы сокращения сроков функциональной верификации – автоматизации процесса создания тестовых окружений.

Автономная верификация СФ-блоков – основной способ проверки будущей системы. Лучшая наблюдаемость, контролируемость состояния блока и скорость моделирования делают автономные окружения наиболее быстрым и эффективным способом проверки функциональности блоков. Анализ эффективности системы на верхнем уровне – задача очень затратная по времени и требующая наличия рабочей модели всей системы и, собственно, целевого ПО. Однако она (как и проверка корректности функционирования системы) может быть декомпозирована и выполнена на автономных окружениях. Условием такой декомпозиции является организация процесса передачи ограничений с верхнего уровня на уровень блоков и результатов характеризации отдельных блоков на уровень системы и других блоков (в частности, подсистемы коммутации) [6]. Всё вышеперечисленное не исключает необходимости моделирования на системном уровне, однако смещает центр тяжести задачи всеобъемлющей проверки СФ-блоков именно на уровень автономных окружений.

Таким образом, для полноценной верификации СнК требуется большое число тестовых окружений для всех СФ-блоков (собственных и третьих сторон) и подсистем различного уровня. Для быстрого создания и поддержки десятков тестовых окружений и проведения мероприятий по функциональной и параметрической верификации СФ-блоков в течение периода разработки СнК нужна инфраструктура, поддерживающая все типы устройств, входящих в СнК. Для создания такой инфраструктуры в первую очередь необходимо определить полный набор требований к тестовым окружениям, определяемый особенностями верифицируемых СФ-блоков, а также стратегией и методиками верификации, применяемыми при верификации блоков различного типа. В следующих главах будет проведен анализ возможных свойств блоков и подходов к их верификации.

II. Классификация СФ-блоков и методик их верификации

Процессорные ядра. Штатный способ их верификации — запуск последовательности команд (управляющей программы, чаще всего генерируемой [7]) непосредственно на модели устройства. Внешнее управление устройством осуществляется через команды отладочного интерфейса (посредством JTAG). Регистровое управление через внутрикристальные интерфейсы (типа AHB) – минимальное или отсутствует вовсе, т.е. разработка программного драйвера, исполняемого на другом CPU, не требуется. Процессоры – инициаторы запросов, поэтому в окружении требуется модель ведомого устройства, отрабатывающего запросы от процессора. Минимальная функциональность такой модели должна включать в себя память (в которую загружается программа) и данные. При верификации процессорных ядер используются эталонные модели различной точности, позволяющие исполнять ту же программу, что и на верифицируемой модели. Для упрощения процесса отладки программ, создаваемых на высокоуровневых языках, в окружении требуются средства тестовой печати (реализация функции printf или её аналогов) и средства обеспечения быстрых операций с памятью для ускорения времени моделирования. Важным моментом для ускорения отладки ПО является возможность использования штатных средств отладки и профилирования ПО (типа gdb) при работе с тестовым окружением процессорного ядра.

Сопроцессоры, автономные сопроцессоры (в том числе ПДП). Требования к тестовым окружениям устройств данного класса схожи с процессорными окружениями, однако их особенностью является необходимость регистрового управления и разработки программного драйвера устройства**.**

Аппаратные ускорители алгоритмов. Основное отличие от сопроцессоров заключается в том, что как эталонная программа, так и тестовый сценарий представляют собой реализации одного и того же комплексного алгоритма, выполненные, как правило, на высокоуровневых языках программирования. В зависимости от модели управления устройством сложность программного драйвера может варьироваться.

Периферийные СФ-блоки. Основная особенность окружений – наличие компонента, имитирующего периферийное устройство (VIP). Программный драйвер для таких блоков может быть достаточно сложным. Кроме того, имеется необходимость синхронизации настроек и действий тестового сценария и активности внешнего VIP. Чаще всего периферийные блоки наравне с процессорами являются инициаторами запросов на системную шину.

Коммутаторы, подсистема памяти. На верхнем уровне абстракции данные устройства являются просто трактом передачи данных. Драйвер (если он требуется) является достаточно простым, а все основные тестовые воздействия задаются трафиком транзакций, для которого требуются специальные генераторы в составе тестового окружения.

Специализированные блоки, не имеющие регистрового управления через стандартные интерфейсы. Примерами таких блоков служат подблоки вычислительного ядра, такие как АЛУ или программный декодер. Основная особенность – отсутствие программного драйвера устройства в явном виде и использование нестандартных интерфейсов для управления и обмена данными. Использование какого-либо общего шаблона для таких блоков затруднено (что не исключает создания узкоспециализированных шаблонов окружений, например для составных блоков АЛУ).

Подсистема, система. В роли системы может выступать не только модель СнК, но и какая либо ее часть, в которой в общем случае возможно наличие всех перечисленных выше устройств. Поэтому требования к окружению системы объединяют в себе требования к окружению процессора и периферийной логики. Акценты верификации на данном уровне делаются на проверку интеграции блоков и эффективности их взаимодействия. Окружение системы, как правило, содержит в себе и окружения подблоков, которые могут работать в пассивном режиме монитора, фоновой проверки или в частично активном режиме (например, включенный имитатор периферийного устройства).

При описании основных особенностей верификации разных классов устройств кроме требований к составу окружения приводились такие параметры, как сложность программного драйвера и источник тестовых воздействий. Далее эти моменты будут рассмотрены уже с точки зрения основных способов создания тестовых сценариев.

Функциональные тесты на языках верификации аппаратуры. Наиболее распространенный вариант – UVM последовательности. Проблемой данного подхода является трудоемкость переработки таких последовательностей для их запуска на модели вычислительного ядра, некоторых вариантах прототипов, на финальных стадиях ко-верификации ПО и аппаратуры, анализа эффективности системы и готовых микросхемах, предполагающих запуск ПО на модели СнК или прототипе/эмуляторе [8]. Поэтому данный способ невыгоден для верификации устройств, требующих разработки сложного программного драйвера, поскольку потребует переработки большого объёма кода при переходе к более поздним этапам верификации блока. Однако языки верификации аппаратуры предоставляют широкие возможности для генерации случайного потока транзакций (например, нагрузки на коммутатор) или потока команд для АЛУ с контролем ограничений и покрытия, что делает их удобными для верификации специализированных устройств и коммутаторов.

Тестовые программы, исполняемые на модели вычислительного ядра. Тестовые программы, написанные на языках ассемблера или языках программирования, компилируемых в базис целевой процессорной архитектуры и исполняемых на модели самого процессорного ядра. Такие тестовые программы являются наиболее удобными для запуска на прототипе системы или готовой ИС, однако если они написаны без учёта определённых правил, то их использование для автономной верификации или прототипирования отдельных СФ-блоков становится затруднительным или требует использования модели процессора, что усложняет и замедляет автономные тестовые окружения.

Тестовые программы, исполняемые «реальным» процессором. Подразумеваются программы, которые управляют автономным тестовым окружением и верифицируемым СФ-блоком, исполняясь на вычислительном ядре рабочей станции или процессоре, входящем в состав платформы прототипирования, но не являющимся частью верифицируемого устройства. Данный способ задания тестовых воздействий является не только быстрым, но и наиболее удобным с точки зрения повторного использования наработанного кода. Так, программный драйвер, отлаженный на автономном окружении, управляемом процессором рабочей станции, и выполненный по рекомендациям, приведенным в [8], легко переносится не только на окружения верхнего уровня СнК в качестве тестовой программы исполняемой моделью вычислительного ядра, но и на уровень серийных образцов ИС в качестве сопровождающего серийную ИС ПО.

Функциональные тесты на основе сценариев или графов. Тестовые сценарии, явным образом не привязанные к языкам программирования или верификации, создаваемые в виде сценариев или графов в определённом формате (например, текстовый или xml) [9]. Для подачи воздействий на устройство требуется средство считывания и исполнения теста. В зарубежных источниках встречаются примеры окружений, в которых реализуются считыватели для воспроизведения сценария и на языке верификации аппаратуры, и в виде тестовой программы, исполняемой на модели процессора, причём оба считывателя реализуются в рамках одного окружения [10]. Каждый узел графа в таком тесте представляет собой действие, исполняемое либо в виде тестовой программы, либо в виде кода на языке верификации аппаратуры. Подобный подход является достаточно гибким с точки зрения переноса сценария верхнего уровня между различными тестовыми окружениями. Однако если гранулярность действий, являющихся узлами графа или шагами сценария, малая, тогда код управления устройством оказывается слишком фрагментированным и непригодным для финальных стадий верификации, исследования системы и его переноса на стадии запуска на готовой ИС.

Верификационным планом может подразумеваться как разработка функционального или исследовательского теста, запуск которого предполагается только на ограниченном числе этапов моделирования (например, случайные тесты на автономном окружении при симуляции RTL или бенчмарки на основе прикладного кода под ОС Linux на прототипе), так и тестовых сценариев, запуск которых предполагается на большинстве этапов верификации (квалификационный набор тестов для подтверждения работоспособности блока во всех базовых режимах, тестовый сценарий для создания нагрузочных ситуаций, облегчённые бенчмарки для анализа производительности, тесты интеграции и т.п.). В первом случае для создания теста логично выбрать способ создания теста, наиболее удобный для генерации воздействий (к примеру, для коммутатора таковыми являются тесты на языках верификации аппаратуры), во втором случае доминирующим требованием является переносимость кода, и наиболее удобным способом тут будут тестовые программы, исполняемые реальным процессором.

С учётом перечисленных особенностей верификации устройств можно сделать вывод о необходимости поддержать все обозначенные способы подачи тестовых воздействий в рамках создаваемого шаблона тестового окружения. Существующие подходы к генерации окружений покрывают только методики, основанные на тестовых сценариях на базе языков верификации аппаратуры (преимущественно на базе UVM) [11]. Методики верификации на базе тестовых программ в наиболее популярных подходах слабо учитываются, либо переносимость тестового кода между этапами автономной, системной верификации и прототипирования в них сильно затруднена. Также упускается из виду проблема взаимодействия процессорных тестов с компонентами тестового окружения, подразумевающая настройку, запуск, останов и синхронизацию разнородных тестовых последовательностей, взаимодействие с которыми требуется для создания требуемых тестовых ситуаций [12].

III. Ключевые особенности генерируемых тестовых окружений

Исходя из обозначенных в предыдущей главе особенностей разных классов СФ-блоков и подходов к их верификации, можно сформулировать основные особенности шаблонного окружения, пригодного для создания окружений для большинства классов СФ-блоков.

- Обеспечение возможности запуска любых обозначенных в предыдущей главе типов тестовых сценариев, в том числе в многопоточном гетерогенном режиме (различные сочетания типов параллельно исполняемых сценариев) с обеспечением единого механизма синхронизации и обмена данными.
- Единая для автономных и системных окружений инфраструктура тестовой печати, управления внешними агентами (VIP) и синхронизации между

процессами, доступная всем типам тестовых сценариев.

- Единое адресное пространство, доступное со стороны тестовых сценариев, верифицируемого устройства и агентов быстрого доступа к памяти (таким образом обеспечивается не только переносимость кода, но и единообразие при создании управляющей программы, исполняемой элементами окружения и моделями процессоров тестируемого устройства).
- Интегрированные агент JTAG интерфейса и средства подключения стандартного отладчика (типа gdb).
- Параметризация верификационных компонент (например, по ширине шин адреса и данных).
- Возможность запуска высокоуровневой (C++/TLM) модели устройства.
- Поддержка пассивного режима работы окружения.
- Все агенты интерфейсов содержат мосты для преобразования транзакций к унифицированному типу [5] таким образом, что большинство высокоуровневых компонент окружения (типа анализаторов, модулей печати и т.п.) не привязываются к конкретным интерфейсам и пригодны к использованию в большинстве окружений различных блоков.
- Окружение имеет базовую часть, содержащую стандартные компоненты типа генераторов тактового сигнала и сброса, мониторов прерываний, контроллеров двунаправленных портов; мониторы; анализаторы; базовые тестовые последовательности и прочие компоненты, не привязанные к реализации верифицируемого устройства. Генерируемые окружения создаются как классы, наследующие описанную базовую часть. Это позволяет не только ускорить процесс создания и унифицировать все интерфейсы взаимодействия с окружением, но и позволяет в одной исходной точке дополнять функциональность уже всех созданных окружений.

Отличительным свойством базовой части является интегрированный интерфейс взаимодействия с тестовыми программами, исполняемыми процессором рабочей станции. К базовой части окружения привязана программная компонента интерфейса взаимодействия ПО с окружениями, обеспечивающая работу драйверов устройств и тестовых сценариев на автономных окружениях. Код таких драйверов пригоден для запуска как на модели процессора, так и на изготовленной ИС СнК.

IV. Интеграция с платформой прототипирования

Главным недостатком запуска тестов на RTL модели является время исполнения. Поэтому для увеличения количества запущенных тестов необходима интеграция с платформой прототипирования, скорость моделирования на которой отличается на порядки.

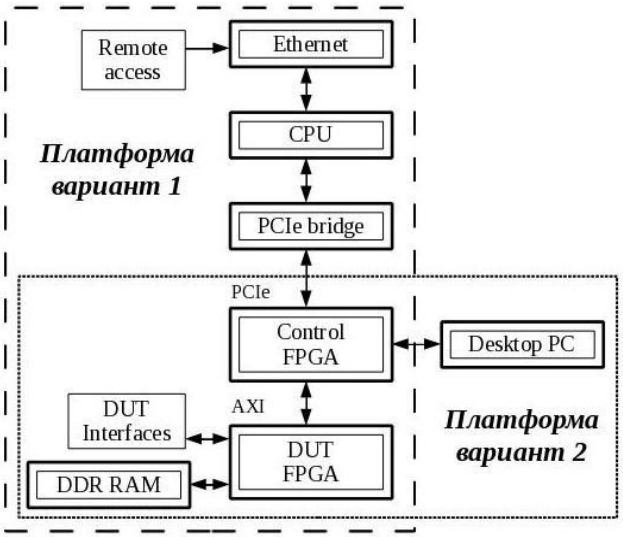

Рис 1. Платформы прототипирования

Рассмотрим ниже возможные варианты платформ прототипирования.

В первом варианте платформы запуск тестовых сценариев осуществляется через удалённый доступ к платформе по каналу Ethernet. Центральный процессор (CPU) через систему аппаратных мостов (PCIe bridge и Control FPGA) имеет доступ к объекту тестирования DUT, погруженному в FPGA (ПЛИС). DUT имеет доступ к памяти DDR RAM и необходимым внешним интерфейсам (DUT interfaces).

Второй вариант представляет собой случай, когда плата прототипирования устанавливается непосредственно в слот PCI настольного компьютера (Desktop PC).

Для ускорения этапа запуска тестов на прототипе необходимо обеспечить их переносимость с этапов автономной верификации. Проблемой тут является то, что на разных этапах верификации взаимодействие с памятью осуществляется различными способами. Например, при работе непосредственно на СнК программа работает в едином адресном пространстве с устройствами, а при работе на прототипе на базе платы, управляемой через PCI, программа должна работать через драйвер для обращения к адресному пространству устройства на прототипе. В последнем случае появляется барьер между двумя адресными пространствами.

С целью обеспечения портируемости тестов на всех этапах верификации необходимо использовать специальный интерфейс для доступа к диапазону памяти, который будет использован устройством во время теста. В программную часть библиотеки, поставляемой с базовым тестовым окружением, включены макросы и реализация интерфейса для работы с памятью, использование которых обязательно

для тестов, которые предполагается переносить между этапами верификации, даже если текущий вариант окружения явным образом не предъявляет особых требований к организации памяти.

В результате появляется возможность запускать тесты, написанные на начальных этапах верификации для автономного окружения, на прототипе без изменений в коде тестов, ускорив таким образом процесс отладки прототипа и введение его в строй.

V. Модульное средство сборки

Подход, использовавшийся авторами ранее, заключался в создании иерархической структуры make-файлов. Данный подход обладал рядом недостатков.

- Отсутствует контроль входных аргументов.
- Не ведется общий журнал действий, выполняемых make-файлом.
- Необходимость дублирования иерархической структуры make-файлов в каждом проекте.
- Затруднено повторное использование кода скриптов сборки тестов и окружений.

Для решения данных проблем требуется инфраструктура, позволяющая упростить перенос тестовых компонент по всем направлениям многомерного пространства этапов верификации [2]. Авторами предлагается использовать модульное средство сборки проектов Project Compiler – инструмент, направленный на автоматизацию маршрута в рамках конкретной группы разработчиков. Суть данной инфраструктуры состоит в использовании для настройки сборщика xml-описаний параметров и команд, что существенно упрощает процесс разработки и верификации. Таким образом, Project Compiler является оболочкой для выполнения команд, автоматизирующей маршрут верификации СнК. Структура данного инструмента представлена на рисунке ниже. Класс Compiler является основным классом, выполняющим загрузку xml и запуск команд. Объект имеет древовидную иерархию, соответствующую иерархии конфигурационных xml-описаний.

Структура конфигурационных xml-файлов представляет собой три группы: файлы для конкретного проекта, файлы для группы проектов и общие (глобальные). Все файлы для конкретного проекта (равно как и файлы для группы проектов) подключаются в одном файле. Параметры и переменные, определённые в глобальных xml, подключаются в xml верхнего уровня группы проектов, который, в свою очередь, подключается в xml верхнего уровня конкретного проекта. Таким образом, части одной команды могут быть описаны в разных файлах. Такая структура позволяет выделить настройку или определение только тех переменных, параметров и команд, которые необходимо настроить для конкретного проекта, что позволяет увеличить повторное использование кода сборки и запуска библиотек, тестов и драйверов.

Данный инструмент поддерживает возможность компиляции нескольких библиотек и компиляции тестов одновременно для разных библиотек и разных целей, например для ядер различной архитектуры. Эти действия также выполняются при помощи конфигурационных xml-файлов.

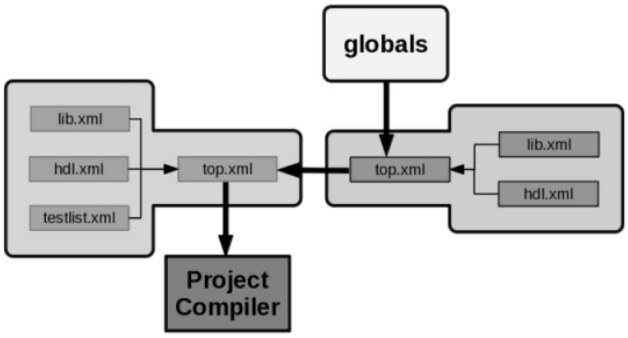

Рис 2. Структура конфигурационных xml-файлов

Модульный подход подразумевает выделение блоков кода, общих для нескольких проектов. Это упрощает процесс разработки, но увеличивает зависимость группы разработчиков от различных коррекций в общей для них части кода. Для работы над проектами с большой иерархической вложенностью, а также для увеличения повторного использования большого объема общего для многих проектов кода, правки в котором влияют сразу на группу проектов, возникла необходимость в разработке системы релизов. Таким образом, было принято решение о необходимости создания системы управления релизами, учитывающей потребности разработчика, верификатора и тополога. Релиз – это срез состояния RTL и верификационных компонент, представляющий собой стабильную версию проекта. Каждый релиз содержит информацию о степени его готовности и особенностях данной версии. Релизы проекта являются опорными точками в его разработке. Это даёт возможность использовать заведомо работающую версию блока при работе над проектом. Система релизов также позволяет выполнять моделирование проекта с заданными пользователем версиями блоков, например для анализа причин сбоев. Для определения статуса релиза, а также для его подтверждения, применяются так называемые квалификационные тесты – при попытке выпуска очередного релиза автоматически выполняется некоторый набор тестов, и в случае его успешного прохождения создаётся новый релиз. Релиз хранится в виде иерархического xml-описания блока и его зависимостей. В xml-описании указывается путь к репозиторию и ревизия данного блока. Основными компонентами описания зависимых блоков являются путь к конкретному блоку в репозитории и его ревизия, которая включена в данный релиз. Данная система релизов отвечает требованиям, которые

предъявляются к ней со стороны разработчика RTL и верификатора, среди которых:

- создавать проекты с иерархической структурой из релизов;
- обеспечить возможность выбора конкретной версии для каждого используемого релиза блока;
- разрешать конфликты при интеграции релизов блоков (СФ-блоки ссылаются на различные релизы своих зависимых блоков);
- подтвердить статус релиза, выпущенного разработчиком, проверив его набором квалификационных тестов.

Использование релизов позволяет минимизировать вероятность случайной правкой сделать неработоспособной модель СнК и систематизировать библиотеку СФ-блоков, в том числе автоматически ведя учёт их использования в разных проектах.

Таким образом, предлагаемый инструмент является в первую очередь средством автоматизации маршрута верификации. Его основным преимуществом является модульность, что позволяет существенно снизить временные затраты на разработку и, как следствие, удешевить конечный продукт. Важно помнить о том, что снижение временных затрат зависит от конкретной группы разработчиков, инфраструктуры, используемой на предприятии, и конкретного проекта. Экономия времени зависит от модульности конкретного проекта и пропорциональна количеству уровней иерархии проекта и повторно используемых модулей. Кроме того, данный инструмент делает процесс разработки и отладки более контролируемым – это достигается за счет подробной системы логирования и контроля входных аргументов. Поддержка системы релизов позволяет разработчику избавиться от зависимости от правок в используемых им блоках и работать со стабильными, прошедшими проверку версиями.

VI. Тестовое окружение как платформа для исследования

Декомпозиция задачи оценки эффективности системы и ее валидации требует средств задания ограничений для исследовательских тестовых сценариев и характеризации отдельных СФ-блоков, а также средств анализа результатов, полученных на автономном уровне, и их передачи на верхний уровень для интегрального анализа характеристик системы.

Для решения данных задач в активные верификационные компоненты, входящие в состав базового окружения, заложены возможности для исследования характеристик блоков (например, время исполнения прикладных задач) в условиях варьирования времён реакции ведомых устройств на активность исследуемого блока и характеристик входного для исследуемого СФ-блока трафика. В свою очередь, в пассивные компоненты, такие как мониторы, интегрирован механизм сбора статистики и сохранения ее в унифицированном формате для последующего анализа. К набору средств автоматизации добавлен модуль для визуализации статистики и автоматизации анализа результатов исследования СФ-блоков.

Основной особенностью инфраструктуры, создаваемой для исследования СФ-блоков, является унифицированность тестовых окружений и, соответственно, выходной информации, получаемой после запуска тестов. Это даёт возможность, во-первых, создавать ПО для анализа результатов тестов, применимое для всех окружений, созданных на базе шаблонного; во-вторых, обеспечить быстрый обмен информацией между окружениями различных блоков и системы, в том числе используя в качестве ограничений для запуска тестов на одном окружении результаты моделирования, полученные на окружении другого блока.

VII. Заключение

Совокупность описанных в статье решений, применённых при разработке инфраструктуры для верификации и исследования СФ-блоков, позволила добиться следующих результатов.

- За счёт автоматизации процесса создания верификационной инфраструктуры ускорить и упростить процесс создания окружений для СФ-блоков, минимизировать процесс адаптации разработчика тестов к окружению нового СФ-блока и упростить перенос тестов между окружениями однотипных блоков.
- Благодаря возможности исполнения тестовых программ на автономных окружениях СФ-блоков, обеспечить переносимость тестов между автономными и системными окружениями.
- За счёт унификации выходной информации, генерируемой тестовыми окружениями, создать единую инфраструктуру для исследования результатов тестов и характеристик СФ-блоков.
- Обеспечить централизацию знаний таким образом, что все новые исправления, улучшения и утилиты автоматически становятся доступны всем, использующим тестовые окружения, созданные на базе шаблонного, что упрощает подержание верификационной инфраструктуры.
- Обеспечить быстрое создание тестовой обвязки для окружений системного уровня (СнК) на базе использования отлаженного кода автономных окружений СФ-блоков, в том числе используемых в целях имитации периферийных устройств, а также использование кода тестов СФ-блоков, отлаженных на автономном уровне, в качестве интеграционных и исследовательских тестов на системном уровне.

Наработанный код тестовых программ и драйверов является легко переносимым между этапами верификации и может быть использован в качестве опорного для разработки ПО для серийных образцов

ИС (требования для обеспечения переносимости в части использования специального интерфейса для доступа к памяти жёстче, чем для драйверов ОС, т.е. при переносе тестовых драйверов на уровень ОС проблем с организаций доступа к памяти не возникает).

Разработанное модульное средство автоматизации маршрута создания и верификации систем на кристалле и система релизов позволили стандартизировать взаимодействие групп разработчиков, верификаторов, топологов и др. в рамках предприятия, а также формализовать и автоматизировать многие этапы проектирования и верификации систем на кристалле, что сказалось на сроках выполнения проектов.

ЛИТЕРАТУРА

[1] Tom A. Are Verification and Validation Different? Does Anyone Care? [Electronic resource] // URL: http://www10.edacafe.com/blogs/thebrekertrekker/2015/08/25/are-verification-and-validation-different-does-anyone-care (дата обращения: 28.03.2016)

[2] К.А. Жезлов. Автоматизация верификации СНК на основе платформенного подхода / К.А. Жезлов, Я.С. Колбасов, А.В. Николаев, Ф.М. Путря, С.Е. Фролова // Электронные компоненты. ЭК1, 2016, с 30-35

[3] Безгласная К.А. Анализ и решение проблем реализации платформенного подхода к построению верификационной инфраструктуры для СнК и СФ-блоков / Безгласная К.А., Колбасов Я.С., Медведев И.А., Путря Ф.М. // Проблемы разработки перспективных микро- и наноэлектронных систем, 2014. Сб. Тр. / под общ.ред. академика РАН А.Л. Стемпковского. М.: ИППМ РАН, 2014. Часть II. С. 69-74.

[4] Головина Е., Макеева М., Николаев А.В., Путря Ф.М., Смирнов А.А. Метод создания и отладки комплексных тестов для функциональной верификации СнК, ориентированный на их повторное использование на всех этапах проектирования // Проблемы разработки перспективных микро- и наноэлектронных систем. Сб. Тр. / под общ.ред. академика РАН А.Л. Стемпковского. М.: ИППМ РАН. Часть II. С. 45-50, 2014

[5] Медведев И.А., Путря Ф.М. Методика верификации межсоединений на базе унифицированной тестовой инфраструктуры // Проблемы разработки перспективных микро- и наноэлектронных систем - 2014. Сб. Тр. / под общ. ред. академика РАН А.Л. Стемпковского. М.: ИППМ РАН, 2014. Часть II. С. 85-90.

[6] Requirements-driven Verification Methodology (for Standards Compliance) [Electronic resource] // www.accellera.orq (дата обращения: 28.03.2016)

[7] А.Н. Мешков. РАЗВИТИЕ СРЕДСТВ ВЕРИФИКАЦИИ МИКРОПРОЦЕССОРА «ЭЛЬБРУС», / А.Н. Мешков, М. П. Рыжов, В. А. Шмелев // Вопросы радиоэлектроники, серия «Электронная вычислительная техника», Выпуск 3, 2014.04.09

[8] Putrya F., Method of free C++ code migration between SoC level tests and stand-alone IP-Core UVM environments, Design & Test Symposium (EWDTS), 2014 East-West,2014, pp.1-4

[9] Verification Academy at DAC2015. Boosting Test-Creation Productivity with Portable Stimulus. [Electronic resource] // verificationacademy.com

[10] TrekSoc Tool [Electronic resource] // http://www.brekersystems.com (дата обращения: 28.03.2016)

[11] Butts S. Are you Still Building Test Benches? / Butts S. // CDNLive, Silicon Valley, 2013.

[12] Jin-kwon P., Cheon-su L., Jae-shin L., Min-jong L. System On Chip (SoC) Device Verification System Using Memory Interface // U.S. Patent US8239708 B2. May 29.

Automation of verification environments development process providing a through design flow for design, verification and research of IP-blocks and SoC

K.A. Zhezlov, Y.S. Kolbasov, A.O. Kozlov, F.M. Putrya, S.E. Frolova

R&D Center «ELVEES», OJSC fputrya@elvces.com

***Keywords* — SoC, verification, verification environments, generation, standardization, parametric control.**

ABSTRACT

The article pays main attention to the automation of the process of infrastructure creation for functional verification of IP blocks, first of all verification environments, compile tools and test results analysis. The attention is also paid to estimating efficiency of a block or system performance. A distinctive feature of the infrastructure being developed is portability of tests and high degree of reuse of environment components, such as tools for compilation of tests, verification environments and test results analysis.

Automated process of creating verification infrastructure accelerated and simplified the process of creating environments for IP blocks. Tests for standalone environments could be run on the system environments. Standardization of verification environments output information makes a single infrastructure for study of test results and characteristics of IP blocks.

The authors describe how to ensure the rapid creation of a test harness for the system-level environments based on the use of debugged code of standalone environments

It is shown that the accumulated code of test programs and drivers is easily portable between the verification steps and can be used as a reference for the development of software for serial IC designs.

The modular tool for automation of design flow for creation and verification of SoC and a system of releases allow to standardize interaction of groups of verificators and developers, which affects the duration of projects. Also that tool allows us to formalize and automate many stages of the design and verification of system-on-chip.

REFERENCES

[1] Tom A. Are Verification and Validation Different? Does Anyone Care? [Electronic resource] // URL: http://www10.edacafe.com/blogs/thebrekertrekker/2015/08/25/are-verification-and-validation-different-does-anyone-care (accessed at: 28.03.2016)

[2] K.A. Zhezlov. Avtomatizacija verifikacii SNK na osnove platformennogo podhoda / K.A. Zhezlov, Ja.S. Kolbasov, A.V. Nikolaev, F.M. Putrja, S.E. Frolova // Jelektronnye komponenty. JeK1, 2016, pp. 30-35

[3] Bezglasnaja K.A. Analiz i reshenie problem realizacii platformennogo podhoda k postroeniju verifikacionnoj infrastruktury dlja SnK i SF-blokov / Bezglasnaja K.A., Kolbasov Ja.S., Medvedev I.A., Putrja F.M. // Problemy razrabotki perspektivnyh mikro- i nanojelektronnyh sistem, 2014. Sb. Tr. / pod obshh.red. akademika RAN A.L. Stempkovskogo. Moscow, IPPM RAN, 2014. Part II. pp. 69-74.

[4] Golovina E., Makeeva M., Nikolaev A.V., Putrja F.M., Smirnov A.A. Metod sozdanija i otladki kompleksnyh testov dlja funkcional'noj verifikacii SnK, orientirovannyj na ih povtornoe ispol'zovanie na vseh jetapah proektirovanija // Problemy razrabotki perspektivnyh mikro- i nanojelektronnyh sistem. Sb. Tr. / pod obshh.red. akademika RAN A.L. Stempkovskogo. Moscow, IPPM RAN, 2014. Part II. pp. 45-50.

[5] Medvedev I.A., Putrja F.M. Metodika verifikacii mezhsoedinenij na baze unificirovannoj testovoj infrastruktury // Problemy razrabotki perspektivnyh mikro- i nanojelektronnyh sistem. 2014. Sb. Tr. / pod obshh. red. akademika RAN A.L. Stempkovskogo. Moscow, IPPM RAN, 2014. Part II. pp. 85-90.

[6] Requirements-driven Verification Methodology (for Standards Compliance) [Electronic resource] // www.accellera.orq (accessed at: 28.03.2016)

[7] A.N. Meshkov, M. P. Ryzhov, V. A. Shmelev (ZAO «MCST») RAZVITIE SREDSTV VERIFIKACII MIKROPROCESSORA «JeL"BRUS»

[8] Putrya F., Method of free C++ code migration between SoC level tests and stand-alone IP-Core UVM environments, Design & Test Symposium (EWDTS), 2014 East-West, 2014, pp.1-4

[9] Verification Academy at DAC2015. Boosting Test-Creation Productivity with Portable Stimulus. [Electronic resource] // verificationacademy.com (accessed at: 28.03.2016)

[10] TrekSoc Tool [Electronic resource] // http://www.brekersystems.com (accessed at: 28.03.2016)

[11] Butts S. Are you Still Building Test Benches? / Butts S. // CDNLive, Silicon Valley, 2013.

[12] Jin-kwon P., Cheon-su L., Jae-shin L., Min-jong L. System On Chip (SoC) Device Verification System Using Memory Interface // U.S. Patent US8239708 B2. May 29.