УДК 621.3.049.771.14 DOI 10.31114/2078-7707-2018-2-79-85

Методы обеспечения переносимости тестовых сценариев между различными верификационными окружениями

А.В. Андрианов

ЗАО НТЦ “Модуль”, г. Москва, aerodronich@yandex.ru

***Аннотация* — При разработке и верификации IP блоков встает задача переноса тестовых сценариев между различными окружениями для проверки корректности работы блока на всех стадиях маршрута проектирования. В статье описаны методы, позволяющие обеспечить переносимость тестовых сценариев между различными верификационными окружениями, такими как: верификационное окружение IP блока, ПЛИС-прототип (под управлением linux и без операционной системы), RTL модель СБИС и реальная микросхема.**

***Ключевые слова* — верификация, СБИС, переносимость, тестовое окружение.**

I. Введение

При разработке IP блока и последующей интеграции в СБИС необходимо контролировать корректность его работы на всех этапах проектирования. Некоторые тестовые последовательности работы с блоком являются схожими для различных окружений и при соблюдении некоторых правил могут быть перенесены между различными тестовыми окружениями без каких-либо изменений.

Возможность переноса тестовых сценариев между различными окружениями может существенно сократить время верификации. В первую очередь, это может достигаться за счет исключения необходимости повторной разработки тестовых сценариев под новое окружение. Во-вторых, возможность переноса тестовых сценариев между окружениями упрощает отладку проблем и исключает случаи внесения дополнительных ошибок при ручном переносе тестового сценария, вызвавшего проблему, между окружениями.

В данной статье будет описан ряд методов, которые позволяют обеспечить переносимость тестовых сценариев между окружениями, начиная от самых простых (на уровне программных интерфейсов) и заканчивая гибридным верификационным окружением.

II. Верификационные окружения

В этой части статьи приведено краткое описание типичных верификационных окружений, использующихся на разных этапах проектирования СБИС. Так как ошибки могут быть выявлены на различных этапах разработки, возможность быстрого переноса между указанными окружениями может позволить быстро определить причину проблемы (например, ошибка в логике работы блока, интеграции блока или настройке окружения).

A. Верификационное окружение IP блока

С этого окружения начинается разработка нового IP блока. В зависимости от сложности блока и наличия внешних интерфейсов это окружение может содержать помимо самого IP блока интерфейс системной шины, поведенческую модель памяти. Если это интерфейс передачи данных, то какую-либо ответную часть.

В данном окружении тесты обычно разрабатываются на языках Verilog/SystemVerilog, иногда с применением методологии верификации uvm. Основная цель верификации на данной стадии – проверка корректности работы блока.

В силу отсутствия процессорного ядра использование программных тестов на языках C/C++, описывающих обращение к регистрам устройства, затруднено. Именно в этом окружении необходимо воспроизводить ошибки, связанные с логикой работы разрабатываемого устройства, в случае их обнаружения на более поздних фазах разработки СБИС.

B. RTL модель СБИС

В данном окружении, как правило, присутствует одно или несколько процессорных ядер. На этом этапе возможна разработка тестового сценария на языке Assembler/C/C++, моделирующего работу с блоком. На данном этапе важно, в первую очередь, проверить корректность интеграции IP блока в состав СБИС. Помимо этого, важна проверка возможности доступа DMA контроллером блока (если он имеется) во все необходимые области памяти, правильность подключения к системной шине, корректность подключения к буферам ввода-выода микросхемы, и т.п. Также желательны: проверка типичных сценариев работы с блоком, производительности, тест на максимальное энергопотребление.

Схематично это окружение представлено на рис. 1. Для наглядности рассмотрен случай, когда верифицируемое устройство и процессорное ядро находятся на разных системных шинах, а в системе присутствует несколько блоков памяти, подключенных

к разным системным шинам. В виде примера таких СБИС можно привести отечественную СБИС К1879ХБ1Я [1] и СБИС 1888ТХ018 [2]

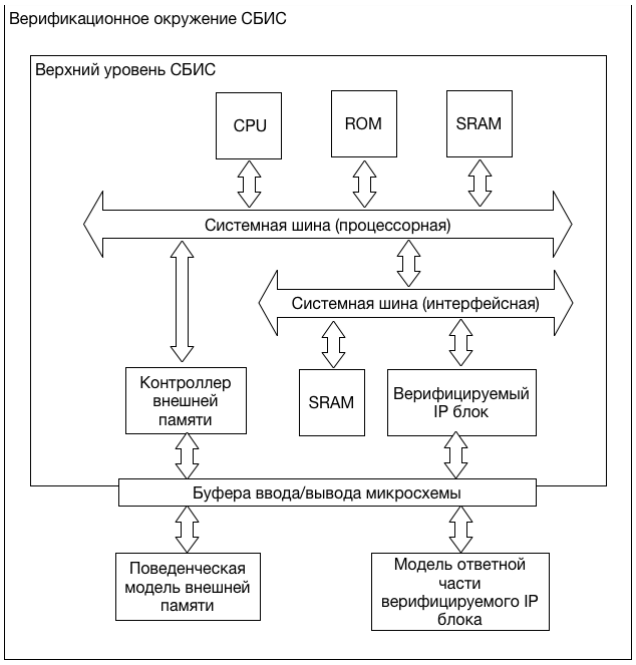

Рис. 1. Упрощенная схема верификационного окружения СБИС

C. ПЛИС-прототип

При разработке некоторых типов периферийных устройств трудно обойтись без фазы ПЛИС-прототипирования. ПЛИС-прототип позволяет проверить работу разрабатываемого блока с реальными ответными устройствами, а также разработать полноценный драйвер для операционной системы, которая будет использоваться в дальнейшем на готовой микросхеме. При этом набор библиотечных элементов, таких как буфера ввода-вывода, макроблоки памяти и системная шина, могут отличаться от тех, что используются в реальной микросхеме или в верификационном окружении IP блока.

D. Реальная микросхема

На данном этапе важна возможность быстрого переноса и запуска имеющихся тестов, позволяющая проверить работоспособность первых образцов микросхемы. Помимо этого тестовые сценарии могут служить готовыми примерами для разработчиков программного обеспечения.

III. Методы обеспечения переносимости тестовых сценариев

A. Обеспечение переносимости на уровне программных интерфейсов (API)

Это один из самых очевидных и простых в реализации методов обеспечения переносимости между окружениями. Однако он требует более тщательного проектирования программного кода для работы с блоком и сам по себе не может обеспечить переносимости в тестовое окружение IP блока, хотя и существенно упрощает повторное использование кода. Ниже приведены самые основные способы обеспечения переносимости кода на уровне программного интерфейса.

1) Разделение тестового сценария на библиотечную часть и основную

Такой подход позволяет скрыть внутреннюю логику работы с блоком от разработчика и сократить сам тестовый сценарий. Более того, библиотечная часть, описывающая типовые сценарии работы с блоком, может применяться в дальнейшем на итоговой микросхеме, ПЛИС-прототипе и при написании драйверов. Использование системы документирования кода doxygen [3] для библиотечной части также очень желательно.

2) Применение функций-оберток для доступа к регистрам устройства и абстракции над используемым контроллером прерываний.

Использование в коде функций-оберток вместо прямого доступа к регистрам IP блока через указатель языка C/C++ дает ряд преимуществ. Помимо очевидного удобства сбора трассы чтения/записи регистров это позволяет без труда переносить код в окружения, где отсутствует прямой доступ к регистрам блока. Эти обертки можно определить как ‘inline’ функции, тем самым избегая накладных расходов, вызванных генерацией компилятором вызова подпрограммы с сохранением стека. Пример (для компилятора gcc) представлен на Листинге 1.

```
static inline uint32_t ioread32(uint32_t const base_addr) {
    return *((volatile uint32_t*)(base_addr));
}
static inline void iowrite32(uint32_t const value, uint32_t const
base_addr) {
    *((volatile uint32_t*)(base_addr)) = value;
}
```

Листинг 1. Пример функций-оберток для доступа к регистрам устройств

Слой абстракции над контроллером прерываний будет также крайне удобен, так как используемые контроллеры прерываний в ПЛИС-прототипе и проектируемой СБИС могут различаться.

3) Специализированная библиотека выделения памяти для DMA операций

Одной из самых частых задач при верификации IP блоков, обладающих встроенным контроллером прямого доступа к памяти, является проверка возможности обращения DMA ко всем присутствующим в СБИС блокам памятей. В различных окружениях может присутствовать разное количество регионов памяти, доступных для DMA операций, поэтому желательно применять слой абстракции для управления выделением памяти. Это позволит вносить минимум изменений в исходный код тестового сценария при его переносе.

Прибегать к функции выделения памяти из стандартной библиотеки языка C (например, newlib) нежелательно, т.к.

- аллокатор памяти стандартной библиотеки языка C работает только с одной общей областью памяти (heap),
- даже самая простая реализация в библиотеке языка C newlib требует достаточно много времени при моделировании,
- освобождение выделенной памяти в большинстве тестовых сценариев не требуется.

Для сравнения времени выделения памяти использовалась RTL модель СБИС, содержащая процессорное ядро с архитектурой ARM, работающее на частоте 800 Mhz, SRAM память, и набор периферийных блоков. Динамическое выделение блока памяти в 512 байт, используя реализацию из newlib, занимает около 3800 нс (лучший случай, первый выделяемый блок, фрагментация отсутствует), в то время как упрощенный линейный аллокатор производит выделение памяти за 2000 нс (всегда).

Таким образом, для большинства тестов на RTL модели СБИС достаточно использовать упрощенный аллокатор, который выделяет память линейно из нескольких имеющихся в системе регионов.

Для более сложных задач, где требуется получить максимальную производительность, может возникнуть необходимость воспользоваться специализированным аллокатором памяти [4].

Наличие слоя абстракции над управлением памятью позволит без проблем использовать различные библиотеки для выделения памяти в различных окружениях.

4) Использовать функции-обертки для ожидания при блокирующих операциях

Естественный подход большинства разработчиков – использовать **блокирующий** API для ожидания выполнения длительных операций. Так как на RTL модели СБИС операционная система чаще всего отсутствует, то ожидание отклика аппаратуры в таком окружении производится либо через циклический опрос значения в регистре статуса устройства, либо через ожидание прерывания, которое также может сводиться к опросу общей структуры данных, выступающей здесь примитивом синхронизации.

Упрощенный пример приведен на Листинге 2.

Этот подход может существенно усложнить повторное использование кода при разработке драйверов для операционных систем, так как циклический опрос в большинстве случаев в драйвере недопустим, а во время ожидания отклика оборудования ресурсы процессора используются другими программами.

Решением этой проблемы может стать применение функций-оберток для блокирующих операций, которые впоследствии могут быть легко заменены на функции операционной системы.

Пример переработанного кода из Листинга 2 приведен на Листинге 3. Здесь реализована упрощенная обертка над ожиданием, используя вызов wait_event, идентичный по синтаксису соответствующему вызову ядра OS Linux.

```
void dma_copy(void *to, void *from, size_t size)
{
   iowrite32(from, DMA_FROM);
   iowrite32(to, DMA_TO);
   iowrite32(size, DMA_LENGTH);
   iowrite32(1, DMA_START);
   while(ioread32(DMA_STATUS) &
DMA_STATUS_WORKING) {};
}
```

Листинг 2. Копирование памяти через DMA, блокирующий API

```
#ifndef LINUX
typedef uint32_t wait_queue_head_t;
#define wait_event(wq, condition) \
   while(!condition) { }

#endif

void dma_copy(wait_queue_head_t *wq, void *to, void *from,
size_t size)
{
   iowrite32(from, DMA_FROM);
   iowrite32(to, DMA_TO);
   iowrite32(size, DMA_LENGTH);
   iowrite32(1, DMA_START);
   wait_event(&wq, !ioread32(DMA_STATUS) &
DMA_STATUS_WORKING));
}
```

Листинг 3. Копирование памяти через DMA, блокирующий API с использованием обертки для ожидания

5) Применять неблокирующий программный интерфейс

Функции-обертки для примитивов синхронизации и диспетчеризации процессов могут немного упростить портирование кода, но не помогут, если потребуется обеспечить работу в окружении, ориентированном на асинхронную модель проектирования (asynchronous design pattern). Более того, разные операционные системы имеют разные API и примитивы синхронизации.

Использование неблокирующего API позволяет решить эти проблемы.

В случае неблокирующего API разработчикам драйверов достаточно использовать существующий код, обернув его необходимыми примитивами синхронизации, специфичными для операционной системы. К плюсам неблокирующего API стоит также отнести то, что с его использованием проще создавать тестовые сценарии, задействующие сразу несколько устройств, например для тестов на максимальную

производительность коммутационной среды и максимальное энергопотребление.

На Листинге 4 приведен код с Листинга 2, переписанный в неблокирующем стиле.

```
void dma_copy_start(void *to, void *from, size_t size)
{

   iowrite32(from, DMA_FROM);
   iowrite32(to, DMA_TO);
   iowrite32(size, DMA_LENGTH);
   iowrite32(1, DMA_START);

}

int dma_copy_finished()
{
   return ! (ioread32(DMA_STATUS) &
DMA_STATUS_WORKING);
}

void dma_copy(void *to, void *from, size_t size)
{
   dma_copy_start(to, from, size)
   while (!dma_copy_finished());;
}
```

Листинг 4. Копирование памяти через DMA, неблокирующий API с блокирующей оберткой

B. Перенесение программы генерации тестового воздействия на хост-компьютер (гибридное верификационное окружение)

Тщательное проектирование архитектуры библиотек кода для работы с IP блоками может позволить повторно использовать разработанный код в дальнейшем на ПЛИС-прототипе и реальной микросхеме, однако не позволяет быстро перенести сценарий, вызвавший проблему, в верификационное окружение IP блока.

Для достижения этой цели необходимо либо существенно усложнить верификационное окружение этого блока, добавив процессорное ядро и блоки памяти, воспользоваться решениями на основе эмулятора с открытым исходным кодом QEMU (например, решения QEMU Cosim [6], FEMU[7]), либо применить гибридное верификационное окружение [8].

Гибридное верификационное окружение предполагает, что тестовый сценарий компилируется и исполняется независимо от модели СБИС. В этом случае информация об операциях чтения и записи регистров, системной памяти, а также о произошедших прерываниях, передается по TCP/IP соединению или через unix socket. Схематично это окружение изображено на рис.2. Направление обмена данными изображено в виде пунктирной линии.

На стороне моделирования используется VPI расширение языка verilog, которое обеспечивает чтение транзакций из сетевого сокета, и трансляцию их в тестовое окружение СБИС.

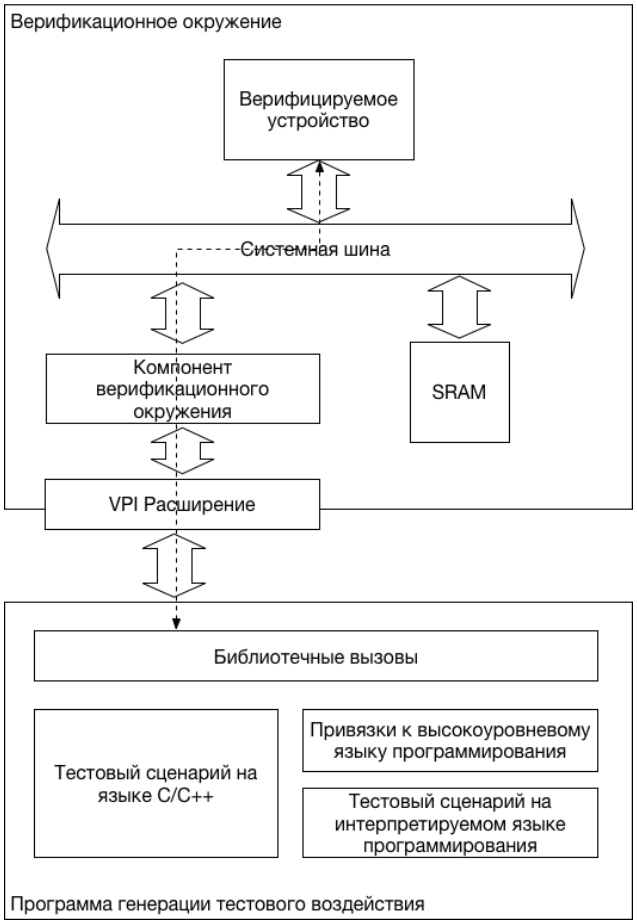

Рис. 2. Схема гибридного верификационного окружения IP блока

Преимущество данного метода в том, что тестовый сценарий представляет собой обычное приложение для ПК, а значит разработчику доступны все современные средства разработки и отладки, такие как пошаговая отладка и трассировка. Более того, многие операции, которые на модели СБИС занимают продолжительное время (форматированный вывод, копирование и заполнение памяти), исполняются локально и работают намного быстрее. Технически также можно реализовать возможность перезапуска сценария без перезапуска моделирования и возможность запуска тестового сценария на другом компьютере, нежели моделирование СБИС.

К недостаткам стоит отнести отсутствие прямого доступа к памяти RTL-модели и необходимость всегда использовать слой абстракции для доступа к ресурсам аппаратуры. Для сохранения возможности переноса на верификационное окружение СБИС необходимо избегать использования в тестовых сценариях функций языков C/C++, которые могут быть недоступны в окружении без операционной системы (например, работа с файловой системой, дескрипторами ввода вывода и т.п.).

Этот подход также можно использовать при верификации СБИС, так как он позволяет существенно упростить отладку тестовых сценариев, а также

верифицировать интерфейсы, дающие доступ в адресное пространство СБИС (например, PCI и PCI Express). Схематично такие верификационные окружения приведены на рис. 3 и 4.

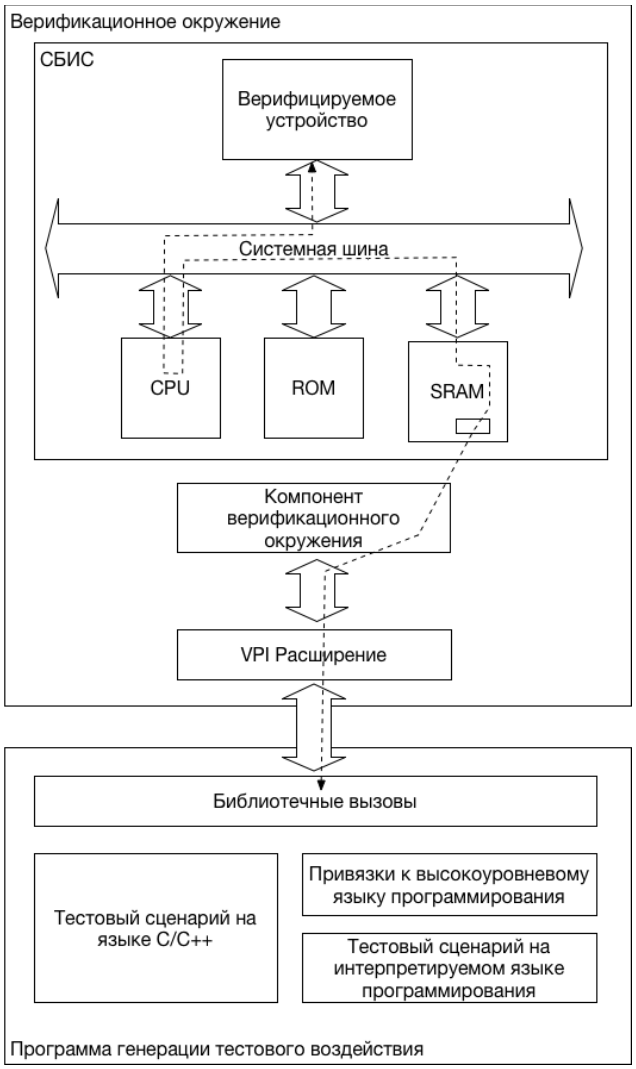

Рис. 3. Схема гибридного верификационного окружения СБИС через внутреннюю память

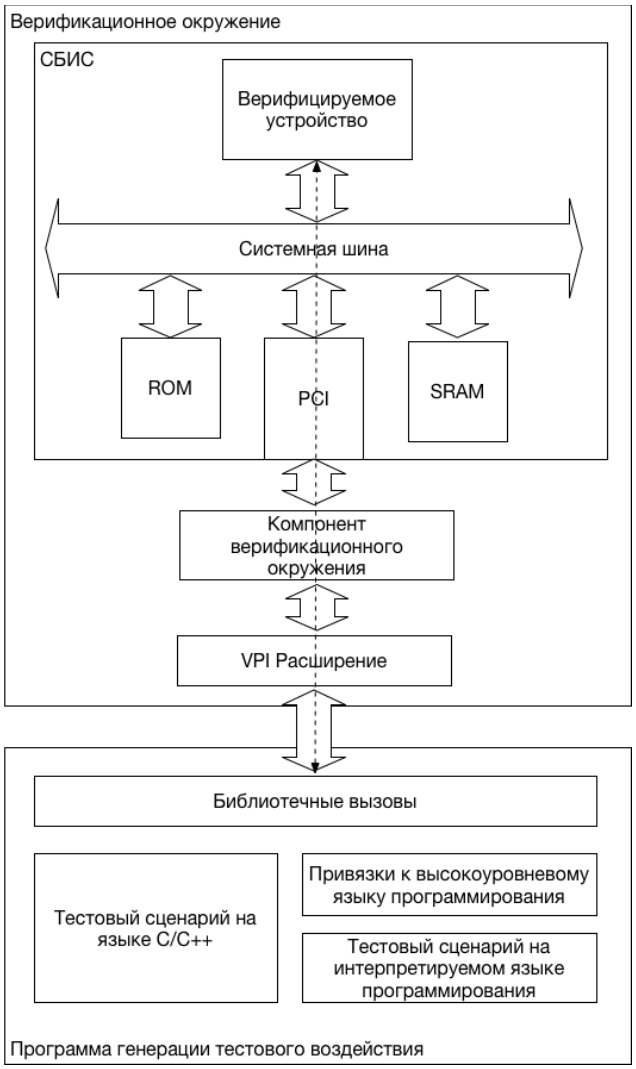

Рис. 4. Схема гибридного верификационного окружения СБИС через интерфейс PCI

К недостаткам такого окружения стоит отнести то, что тестовый сценарий будет скомпилирован для архитектуры хост-компьютера (как правило, это intel x86/x86_64, порядок байт - little endian). Таким образом порядок байт и размеры некоторых базовых типов данных в языках C/C++ могут отличаться.

Решением данной проблемы может служить либо повышение требований к качеству кода, как уже было упомянуто выше, так и использование эмулятора qemu в режиме “usermode emulation”.

Этот режим позволяет запускать программы пространства пользователя linux, скомпилированные под другую архитектуру, как если бы они были скомпилированы для хост-архитектуры.

Такой режим эмуляции требует намного меньше ресурсов, нежели эмуляция всей системы (режим системной эмуляции QEMU, применяемый решениями FEMU/QEMU Cosim) и позволяет компилировать тестовый сценарий в условиях, максимально приближенных к RTL модели СБИС.

C. Применение высокоуровневого скриптового языка для разработки тестовых сценариев

При использовании гибридного верификационного окружения выбор языков программирования для разработки тестового покрытия не ограничивается C/C++. Существует возможность написания тестовых сценариев на высокоуровневых языках, таких как lua, python, perl и любых других.

Применение высокоуровневых языков может существенно сократить временные затраты на написание тестовых сценариев за счет наличия богатого набора библиотек, наличия более гибкого и современного синтаксиса, автоматического управления памятью, а также отсутствием проблем переносимости, связанных с размерами типов данных и порядком байт.

Однако необходимость переноса тестовых сценариев в окружение без операционной системы для верификации готовых микросхем накладывает существенные ограничения на спектр высокоуровневых языков программирования, которые можно задействовать для решения данной задачи. В этой части статьи приведена информация о трех популярных высокоуровневых языках программирования, которые возможно запустить в окружении без операционной системы, и, соответственно, которые могут быть использованы при верификации СБИС.

Lua – это популярный легковесный интерпретируемый язык, ориентированный на встраивание в приложения. Этот язык широко используется в большом количестве проектов от разработки компьютерных игр до прототипирования встраиваемых систем[9] и задач машинного обучения[10]. Интерпретатор языка lua можно скомпилировать и использовать в окружении без операционной системы даже при малом объеме доступной оперативной памяти. Для интерпретатора lua версии 5.1 существует проект lua2c, который дополнительно позволяет транслировать код, написанный на lua, в код на C, использующий вызовы библиотеки интерпретатора lua.

Python – один из наиболее известных современных высокоуровневых языков общего назначения. Его отличают минималистичный синтаксис, богатый набор функций стандартной библиотеки и огромное количество сторонних библиотек. Проект micropython [11] позволяет работать интерпретатору этого языка программирования в окружении без операционной системы, однако поддерживает далеко не полный набор функций стандартной библиотеки. К преимуществам этого языка стоит отнести также наличие расширения myhdl[12], позволяющего разрабатывать и верифицировать синтезируемые цифровые блоки используя python как основной язык для описания аппаратуры и создания тестов.

Javascript (baremetaljs) – хотя изначально основной нишей данного языка программирования была веб-разработка с появлением таких платформ, как node.js, этот язык стал активно применяться в виде языка общего назначения. Проект baremetaljs позволяет использовать данный язык на микроконтроллерах и в окружении без операционной системы, что делает его технически пригодным для использования в виде высокоуровневого языка программирования для написания верификационных сценариев.

IV. Заключение

Представленные в данной статье методы, все вместе или по отдельности, могут позволить разработчикам переносить тестовые сценарии между различными окружениями, сводя к минимуму вероятность внесения дополнительных ошибок, добиваясь повторного использования уже написанного кода на дальнейших этапах проектирования СБИС. Возможность быстрого переноса тестовых сценариев между окружениями может упростить поиск ошибок. Перенос и запуск этих сценариев в верификационном окружении IP блока позволяет учитывать их при расчете суммарного покрытия блока тестами.

Важно отметить, что упомянутые в этой статье методы подходят для верификации периферийных блоков и интерфейсов, однако едва ли будут достаточными для полноценной верификации непосредственно ядра процессора.

Дополнительная возможность применять высокоуровневые языки программирования для верификации IP блоков может ускорить разработку тестовых сценариев.

Литература

[1] «Отечественная СБИС декодера цифрового телевизионного сигнала К1879ХБ1Я (PDF)», Шевченко П.А., Цифровая обработка сигналов и её применение — DSPA'2011, Москва 2011
URL: http://www.module.ru/upload/images/1354523271art_dspa2011_1.pdf (дата обращения 01.04.2016)

[2] И.П. Филимонова, И.В. Безкоровайный, Д.И. Дрягалкин, Г.О. Чумаченко, В.Ю. Залётов, А.В. Андрианов: «Мультимедийная СНК с процессорными ядрами PowerPC и NMC3», Международный форум «Микроэлектроника 2017», Алушта 2017.

[3] URL: http://www.doxygen.org (дата обращения: 08.04.2018)

[4] «Гибридный метод аллокации массивов памяти в аппаратных платформах с разветвленной структурой памяти на базе процессора NeuroMatrix® DSP», С.В. Мушкаев, А.В. Андрианов

[6] URL: https://www.aldec.com/en/solutions/functional_verification/qemu_co_sim (дата обращения 01.04.2016)

[7] “FEMU: A firmware-based emulation framework for SoC verification”, 2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) URL: http://ieeexplore.ieee.org/document/5751510/ (дата обращения 01.04.2016)

[8] “Методика гибридной верификации СБИС “Система-на-Кристалле”, Андрианов А.В., Шагурин И.И. "Датчики и системы" , 2018г., №2, с. 14-18.

[9] URL: http://www.eluaproject.net/ (дата обращения 01.04.2016)

[10] URL: http://torch.ch/ (дата обращения 01.04.2016)

[11] URL: https://micropython.org/ (дата обращения 01.04.2016)

[12] URL: http:// myhdl.org/ (дата обращения 01.04.2016)

Methods of Achieving Test Scenario Portability Between Different Verification Environments

A.V. Andrianov

Research Centre “Module” JSC, Moscow, aerodronich@yandex.ru

Abstract —**During development and verification of IP cores it is often needed to port test scenarios between different environments to check that the IP core works correctly.**

Development and verification of IP cores is done in a set of distinct environments: the IP core testbench, the FPGA-prototype, RTL model of the SoC containing the IP core and finally the real SoC.

These environments differ drastically and porting test scenarios between environments can be very useful during debugging.

The article describes a set of useful methods that can make verification scenarios written in C/C++ or a higher level scripted language become portable between different environments and even be reused later as part of the software development kit for the SoC (SDK).

The very first method of providing portability is a careful design of application programming interfaces (API). Apart from obvious splitting of the test scenarios into a library part and the actual test, using wrapper functions for all register operations, a specialized memory allocation library for managing DMA memory is highly recommended over allocating test data arrays statically. This is especially useful when the SoC has a set of different memory blocks DMA access to which needs to be verified.

Busy-wait loops in code should be either avoided by using a non-blocking API or wrapped in functions, mimicking the operating system scheduling API.

However, careful API abstractions cannot bring the test scenarios written in C/C++ to the verification testbench of the IP core that usually lacks a CPU required to execute them. To avoid complicating the testbench by adding a CPU and memories, or using a QEMU for CPU emulation, a hybrid approach is suggested. In hybrid verification environment the test scenario is executed as a user space process, communicating with the actual simulation over a TCP/IP channel or a unix socket.

This also allows comfortable usage of high-level languages for verification purposes instead of C/C++. To retain portability to the target SoC, languages like python (micropython), lua (elua) and javascript (baremetaljs) are recommended, since they provide a way to run the test scenario in the “bare-metal” environment, without any operating system.

***Keywords* — SoC, verification, verification environment, portability.**

REFERENCES

[1] «Otechestvennaja SBIS dekodera cifrovogo televizionnogo signala K1879HB1Ja (PDF)», (Digital television broadcast devoder IC) Shevchenko P.A. Cifrovaja obrabotka signalov i ejo primenenie — DSPA'2011, Moscow 2011 URL: http://www.module.ru/upload/images/1354523271art_dspa2011_1.pdf (access date: 01.04.2016)

[2] I.P. Filimonova, I.V. Bezkorovajnyj, D.I. Drjagalkin, G.O. Chumachenko, V.Ju. Zaljotov, A.V. Andrianov: «Mul'timedijnaja SNK s processornymi jadrami PowerPC i NMC3», (Multimedia SoC with PowerPC processor cores and NMC3 DSP), Mezhdunarodnyj forum «Mikrojelektronika 2017», Alushta 2017.

[3] URL: http://www.doxygen.org (access date: 08.04.2018)

[4] «Gibridnyj metod allokacii massivov pamjati v apparatnyh platformah s razvetvlennoj strukturoj pamjati na baze processora NeuroMatrix® DSP», (“Hybrid method of array allocation in branched memory organisation platforms based on NeuroMatrix® DSP”) S.V. Mushkaev, A.V. Andrianov

[6] URL: https://www.aldec.com/en/solutions/functional_verification/qemu_co_sim (access date: 01.04.2016)

[7] “FEMU: A firmware-based emulation framework for SoC verification”, 2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) URL: http://ieeexplore.ieee.org/document/5751510/ (access date: 01.04.2016)

[8] “Metodika gibridnoj verifikacii SBIS “Sistema-na-Kristalle”, (Hybrid “System-On-Chip” verification methodology) Andrianov A.V., Shagurin I.I. "Datchiki i sistemy" , 2018., №2, p. 14-18.

[9] URL: http://www.eluaproject.net/ (access date: 01.04.2016)

[10] URL: http://torch.ch/ (access date: 01.04.2016)

[11] URL: https://micropython.org/ (access date: 01.04.2016)

[12] URL: http://myhdl.org/ (access date: 01.04.2016)