УДК *621.3.049.771.14*

Практические способы оптимизации процесса регрессионного тестирования СБИС “СнК”

Андрианов Андрей Владимирович
andrianov@module.ru

ЗАО НТЦ «Модуль»
125190, г. Москва, а/я 166
Тел. +7 (495) 531-30-80
http://www.module.ru

Аннотация: В статье рассмотрен ряд практических способов сокращения временных затрат при разработке и отладке программных тестов СБИС “СнК”, а также при последующем регрессионном тестировании. Рассматриваются вопросы оптимального выбора аппаратного обеспечения, тонкой настройки операционной системы, особенности верификационного окружения и средства пакета ПО Cadence XCELLIUM.

Ключевые слова: СБИС, моделирование, регрессионное тестирование, xcellium, cadence, numa

Practical ways of optimizing regression testing of “system-on-chip” projects

Andrianov Andrew Vladimirovich
andrianov@module.ru

JSC RC «Module»
P.O. Box 166, Moscow, Russia, 125190
http://www.module.ru

Abstract: The article describes a set of practical ways to reduce time costs when developing and debugging test scenarios of “system-on-chip” VLSI and optimize regression testing. The article covers issues of picking optimal hardware for the task, fine-tuning operating system, designing the testbench and using select features from Cadence XCELLIUM software package.

Keywords: SoC, simulation, regression testing, xcellium, cadence, numa.

Введение

В связи с ростом сложности современных СБИС класса “система-на-кристалле” встает вопрос сокращения временных издержек при сборке проекта, разработке тестовых сценариев, а так же при регрессионном тестировании. Сокращение временных затрат на начальную и повторную сборку проекта может существенно ускорить процесс разработки тестовых сценариев и верификационного окружения ввиду итерационности этого процесса. Ускорение регрессионного тестирования дает возможность получать чаще информацию о состоянии регрессии и оперативно исправлять возникающие проблемы.

В типичной СБИС, как правило, присутствует одно или несколько процессорных ядер. Разработка тестовых сценариев для верификации подключения IP блоков СБИС ведется на языках Assembler/C (реже C++), которые выполняют типичные операции для этих блоков. На данном этапе важно, в первую очередь, проверить корректность интеграции IP блока в состав микросхемы. Помимо этого, важна проверка возможности доступа DMA контроллером блока (если он имеется) во все требуемые области памяти, правильность подключения к системной шине, корректность подключения к буферам ввода-вывода микросхемы, и т.п.. Также желательны: проверка типичных сценариев работы с блоком, производительности, тест на максимальное энергопотребление. Для основных процессорных ядер и вспомогательных DSP ядер (если они есть), могут присутствовать дополнительные тестовые сценарии.

Верификационное окружение такой СБИС может содержать ответные части для используемых интерфейсных IP блоков, модели внешней памяти, а также вспомогательные компоненты, отвечающие за отладочную печать, обеспечивающие загрузку и выгрузку данных в память СБИС, остановку моделирования по запросу тестового сценария, интеграцию с системой запуска тестов и т.п.

В статье рассмотрены практические способы оптимизации регрессионного тестирования СБИС, которые вместе или по отдельности позволяют сократить временные издержки и ускорить процесс разработки СБИС. Рассматривается выбор конфигурации оптимального оборудования для типичных сценариев разработки и тестирования, организация системы сборки, а также возможные оптимизации на уровне верификационного окружения.

Система сборки и тестирования

Прежде всего, стоит выделить два различных сценария работы с проектом СБИС, которые существенно различаются по требованиям ко времени сборки проекта:

1. **Разработка тестовых сценариев и компонентов верификационного окружения СБИС**
   В данном случае, важно максимально быстро производить начальную и повторную сборку проекта, а также максимально быстро производить моделирование одиночных тестов. Многопоточная сборка проекта здесь крайне желательна, так как позволяет существенно ускорить процесс.
2. **Пакетный запуск тестов при регрессионном тестировании**
   Здесь время начальной сборки не играет большой роли на фоне общего времени прохождения тестовой регрессии. Повторная сборка проекта здесь, как правило, не производится, чтобы сохранить максимальную воспроизводимость окружения при каждой сборке. Основной выигрыш по времени можно получить при помощи параллельного запуска ряда тестов, сокращения времени прохождения отдельных тестов и распределения нагрузки между несколькими серверами. Многопоточная сборка проекта здесь нежелательна, так как она делает использование ресурсов многопроцессорных систем менее предсказуемым, что усложняет планирование нагрузки на сервера для автоматизированных систем, таких как Jenkins CI.

Система сборки проекта может существенно сократить время первичной и повторной сборки проекта, если будет активно задействовать ресурсы многопроцессорных систем. В ее функции входит сборка проекта, предоставление средств для запуска тестовой регрессии, а также возможность задания описания для компонентов проекта (Список файлов, опции сборки, указание зависимостей между компонентами проекта и т.п.). В зависимости от реализации, система сборки может также задействовать средства САПР для сбора статистической информации и различных метрик (например, покрытия кода и аппаратуры тестами, диагностической информации и т.п.). В настоящее время существует огромное количество различных систем сборки, которые можно использовать для сборки проектов сложных СБИС. Здесь можно использовать как простые сценарии на языках bash, cshell, tcl, gnu make или python, или задействовать более сложные системы сборки, такие как cmake[1], предоставляющие высокоуровневые языки для описания процесса сборки, и содержащие в себе инструменты для организации и запуска тестовой регрессии (такие, как ctest).

Типичный проект СБИС "Система-на-кристалле", состоит из описания аппаратуры на языках Verilog или VHDL, верификационного окружения, а также ряда программных компонентов, таких как тестовые сценарии, VPI и DPI расширения для САПР, использующиеся в верификационном окружении и т.п. На Рисунках 1 и 2 приведены схематично процессы сборки программных (При помощи компилятора GCC) и аппаратных (используя САПР Cadence Incisive/Xcellium) компонентов проекта, соответственно.

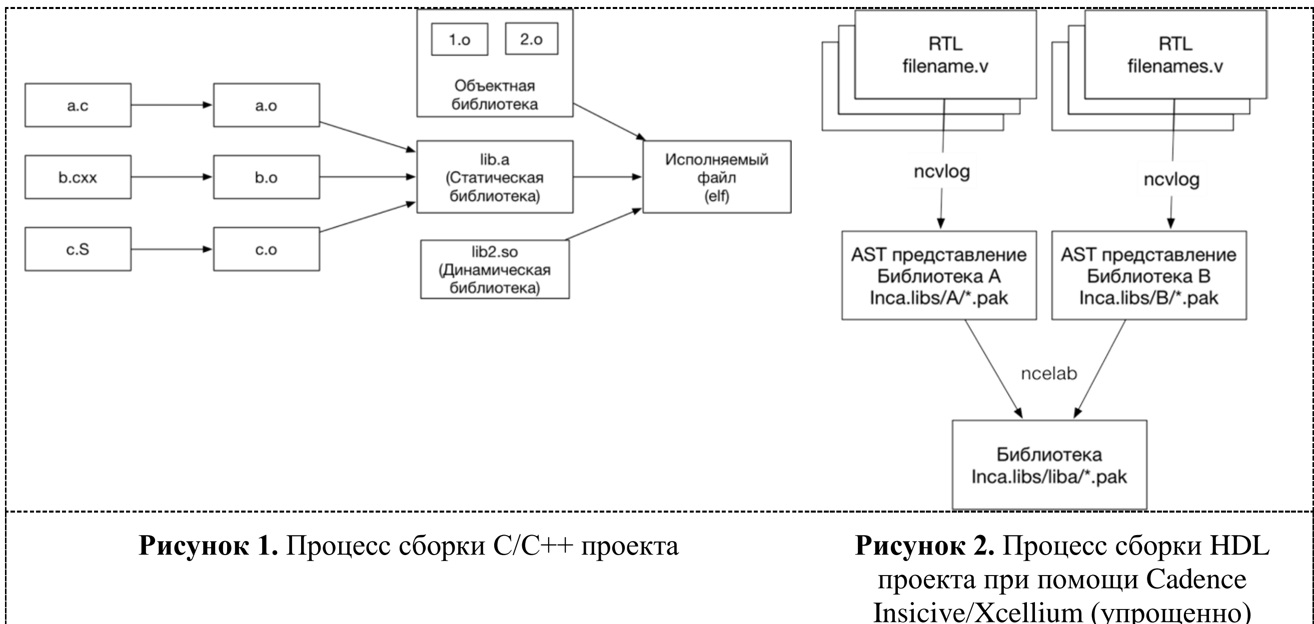

Рисунок 1. Процесс сборки C/C++ проекта

Рисунок 2. Процесс сборки HDL проекта при помощи Cadence Insicive/Xcellium (упрощенно)

Теперь рассмотрим способы, при помощи которых можно сократить время начальной и повторной сборки проекта.

Многопоточная компиляция

Многопоточная компиляция является стандартом для больших программных проектов и хорошо поддерживается большинством современных систем сборки. Использование многопоточной сборки может серьезно уменьшить время сборки программных компонентов проекта. Как видно из Рисунка 1, где представлены схематично этапы сборки программного обеспечения, процесс хорошо параллелизуется на уровне компиляции отдельных файлов. Необходимость повторной компиляции отслеживается системами сборки путем сравнения времени изменения файлов с исходным кодом и результатов компиляции. Задействование многопоточной компиляции совместно с инструментами Cadence Incisive/Xcellium затруднено, из-за архитектуры данного пакета САПР. Как видно из Рисунка 2, инструменты Cadence используют общий файл базы данный для хранения результатов компиляции и элаборации. Для организации доступа к базе данных используется система блокировок. Если процесс ncvlog или ncvhdl, который производит компиляцию файла, попытается получить доступ на запись к библиотеке, используемой другим процессом, то он будет вынужден ожидать, пока другой процесс снимет блокировку. Таким образом, можно ускорить процесс компиляции, используя несколько процессов компилятора, только в случае если параллельно будут компилироваться несколько независимых библиотек, как показано на Рисунке 2.

Функционал MSIE (Multiple snapshot incremental elaboration) САПР Cadence Incisive (Xcellium)

Процесс компиляции verilog и systemverilog не занимает много времени, в отличие от процесса подготовки модели СБИС (элаборации, elaboration). MSIE рекомендуется разработчиком САПР Cadence для сложных СБИС. Этот функционал позволяет снизить время повторной сборки, благодаря проведению повторной элаборации только тех частей проекта, в которые вносились правки. Также при использовании этого функционала, отдельные части модели (на уровне отдельных подсистем или даже отдельных блоков) могут элаборироваться независимо, позволяя это делать в несколько потоков и таким образом, более полно использовать ресурсы многопроцессорных систем. Ввиду того, что результаты данной операции так хранятся в той же базе данных, что и скомпилированные библиотеки, многопоточная элаборация может дать выигрыш только при элаборации независимых частей проекта. В противном случае, один из процессов будет ожидать снятия блокировки с библиотеки, для которой требуется эксклюзивный доступ. Несмотря на это ограничение, многопоточная элаборация может серьезно сократить время сборки проекта СБИС, как видно из Таблицы 1. Для тестирования использовался проект СБИС, содержащий процессорное ядро ARM Cortex A5 и набор периферийных блоков.

	Компиляция	Компиляция и элаборация	Повторная элаборация
Без MSIE	22.84	51.38	17.74
MSIE, 1 поток	22.89	61.37	7.78
MSIE, 8 потоков	20.13	28.34	5.2

Таблица 1. Время (в секундах) компиляции/элаборации проекта СБИС в различных вариантах сборки

Система кэширования результатов компиляции ccache (Compiler Cache) с единой директорией кэширования для всех пользователей

ccache – решение которое позволяет сохранять в кэш на файловой системе результаты компиляции отдельных версий файлов и при повторных сборках автоматически использовать результаты предыдущих сборок, если они доступны.

Использование ccache существенно сокращает время повторной сборки из чистой сборочной директории. Для сравнения производилась компиляция программных тестов СБИС из 1096 файлов на языке C и Assembler и суммарная компоновка 336 исполняемых файлов. Сравнительные результаты начальной и повторной сборки набора тестов СБИС на языке C представлены в Таблице 2.

На серверах рационально использовать единый каталог для хранения данных ccache для совместной работы нескольких пользователей. При небольшом объеме компилируемого кода, их можно разместить в оперативной памяти используя файловую систему tmpfs, что существенно ускорит доступ к кэшированным данным.

	Компиляция без ccache	Компиляция с ccache
1 поток	64.174s	35.612
2 потока	37.658s	16.554

Таблица 2. Время (в секундах) компиляции набора тестов СБИС

Тонкая настройка планировщика при работе на материнских платах с несколькими физическими процессорами

Многие серверные материнские платы содержат разъемы для нескольких физических процессоров. Такие сервера являются системами с неравномерным доступом к памяти (NUMA, Non-Uniform Memory Access). Это значит, что время доступа к нелокальной, относительно данного процессора, памяти может различаться. По умолчанию, планировщик ядра OS Linux будет выделять память в зависимости от доступности, а процесс моделирования может исполняться то одним, то другим процессорным ядром в зависимости от нагрузки.

Это поведение может быть нежелательным, особенно при финальном моделировании СБИС с учетом задержек (postlayout simulation), когда потребляется большое количество памяти. Это можно исправить, воспользовавшись инструментом numactl. Команда numactl позволяет выводить диагностическую информацию о конфигурации системы, а так же принудительно привязывать запускаемые процессы к конкретному физическому процессору и его локальной памяти, что в ряде случаев может дать существенный выигрыш.

Многие современные процессоры при большой нагрузке могут снижать свою тактовую частоту, чтобы избежать перегрева. Таким образом, можно использовать информацию о нагрузке для планирования запуска следующего процесса моделирования на наименее нагруженном в данный момент физическом процессоре.

Требования к аппаратуре сервера

При использовании многопоточной компиляции и элаборации огромное значение имеет производительность дисковой подсистемы. Типичный проект СБИС содержит огромное количество файлов небольшого размера, поэтому при сборке проекта огромную роль играет время случайного доступа.

Устройство хранения данных	Время, с
NVME SSD накопитель	80.427
3.5’ SATA Жесткий диск	171.264
NFS раздел (1Gbps сеть)	92.103
RAID-1 зеркало из двух SAS жестких дисков	89.980

Таблица 3. Время компиляции и элаборации проекта СБИС (в 4 потока) на разных устройствах хранения данных

В Таблице 3 приведено сравнение времени компиляции проекта СБИС при размещении исходных файлов на накопителях с разным временем случайного доступа к файлам. Приведены типичные значения, так как это время может существенно варьироваться в зависимости от конфигурации оборудования и нагрузки.
Уже скомпилированная модель СБИС, напротив, содержится в одном, реже нескольких файлах и при запуске моделирования полностью загружается в оперативную память. Во время моделирования запись на диск производится только для сохранения контрольных точек моделирования, загрузки и выгрузки файлов в модель СБИС и сохранения журнала моделирования. На момент написания данной статьи процесс моделирования при помощи САПР Cadence Incisive и Xcellium происходит в один поток. Поэтому непосредственно на время моделирования будет влиять тактовая частота процессорного ядра, на котором будет выполняться моделирование, но никак не общее количество процессорных ядер в системе.
Из вышеприведенных данных можно заключить, что для разработки тестовых сценариев и верификационного окружения рационально выбирать конфигурацию серверного оборудования с высокопроизводительной дисковой подсистемой и высокой частотой процессора. Для регрессионного же тестирования СБИС производительность дисковой подсистемы и тактовая частота процессора большой роли не играют, а основной выигрыш можно получить за счет большего количества процессорных ядер и, соответственно, одновременных процессов моделирования.

Верификационное окружение

Тщательно спроектированное верификационное окружение может серьезно ускорить разработку и отладку тестовых сценариев. К сожалению, в виду объемности данной темы, в данной статье будут приведены лишь несколько основных приемов, которые, по мнению автора, могут серьезно ускорить разработку и отладку тестов и сократить время прохождения тестовой регрессии.

Некоторые из рассмотренных ниже приемов предполагают наличия канала обмена данными между тестом, исполняемым на моделируемом процессоре, и верификационном окружением. Таким каналом может служить, например, фрагмент памяти в накристальном ОЗУ, один из интерфейсов (например, UART, GPIO) или неиспользуемые компилятором специальные регистры процессора (если они есть). Исполняемый процессором тест передает код произошедшего события и аргументы (если они требуются), а верификационное окружение производит обработку поступающих событий.

Использование прямого доступа к памяти СБИС со стороны верификационного окружения, для сокращения объема передаваемых по каналу обмена данных

Верификационное окружение может иметь доступ к внутренней памяти СБИС через иерархические ссылки, что может существенно сократить объем передаваемых данных между тестом исполняемым на моделируемом процессоре и окружением. Для этого вместо непосредственной передачи всех данных по каналу обмена, достаточно передать адрес ячейки памяти, где расположены данные.

```
__attribute__((no_instrument_function))
__attribute__((optimize("-O0"))) void fast_printf(void
*fmt, ...)
{
	send_event(EVENT_PRINTF,
__builtin_frame_address(0));
}
```

Листинг 1. Пример функции-заглушки, переадресующей вызов printf верификационному окружению

Перенос выполнения “долгих” библиотечных функций языка C на сторону верификационного окружения

Некоторые часто используемые функции стандартной библиотеки языка C, такие как printf, memset, memcpy, memcmp, при моделировании требуют значительного времени даже на небольших объемах данных, хотя и не производят непосредственно тестирования каких бы то ни было компонентов микросхемы. Как правило, они используются для отладочной печати, начальной инициализации, подготовки и анализа эталонных данных.

```
00048784 <fast_printf>:
  48784:   e92d000f   push   {r0, r1, r2, r3}
  48788:   e92d4800   push   {fp, lr}
  4878c:   e28db004   add    fp, sp, #4
  48790:   e1a0300b   mov    r3, fp
  48794:   e1a01003   mov    r1, r3
  48798:   e3a00005   mov    r0, #5
  4879c:   ebffffcf   bl     486e0 <send_event >
  487a0:   e320f000   nop    {0}
  487a4:   e24bd004   sub    sp, fp, #4
  487a8:   e8bd4800   pop    {fp, lr}
  487ac:   e28dd010   add    sp, sp, #16
  487b0:   e12fff1e   bx     lr
```

Листинг 2. Сгенерированный компилятором gcc для ARM код функции из Листинга 1

Типичное время исполнения некоторых библиотечных функций на СБИС с ядром ARM Cortex A5 и набором периферийных блоков приведено в Таблице 4.

Использование таких часто применяемых, но и одновременно “долгих” функций можно перенести на сторону верификационного окружения, и, желательно, сделать это, передав по каналу обмена данными наименьшее количество данных. Приведенный здесь метод основывается на наблюдении, что при вызове функции с переменным количеством аргументов с отключенной оптимизацией, компилятор (gcc) размещает все аргументы функции в стеке, который можно разобрать программно. На Листинге 1 приведен пример вызова функции с переменным количеством аргументов, а на Листинге 2 результат дизассемблирования.

Для компиляции использовался компилятор GNU C Compiler 6.3.0 для архитектуры ARM.

Таким образом, достаточно передать верификационному окружению код события и адрес текущего фрейма стека независимо от количества аргументов. К недостатку такого подхода стоит отнести то, что формат хранения переменных в стеке может различаться в зависимости от используемого ABI, версии компилятора, а также используемой процессорной архитектуры. В случае изменения чего-то из вышеупомянутого потребуется вносить изменения в компонент верификационного окружения, осуществляющий разбор стека.

При необходимости запуска на итоговой микросхеме, достаточно использовать программную реализацию этих функций. Здесь важно отметить, что при этом время исполнения отдельных частей программы будет меняться, вплоть до изменения порядка событий программы, что, в свою очередь, увеличивает требования к качеству исходных кодов тестового сценария.

Операция	Время модели, ns	Реальное время, с
Форматирование строки (sprintf), реализация nano из библиотеки newlib (sprintf(buf, “Hello world %d %d %d\n”, i,j,k))	44688	62
Копирование буфера размером 512 байт при помощи стандартной функции memcpy()	130219	174
Заполнение буфера размером 512 байт нулями, используя функцию memset() библиотеки newlib	103344	139

Таблица 4. Время (в секундах) компиляции набора тестов СБИС

Использование гибридного верификационного окружения

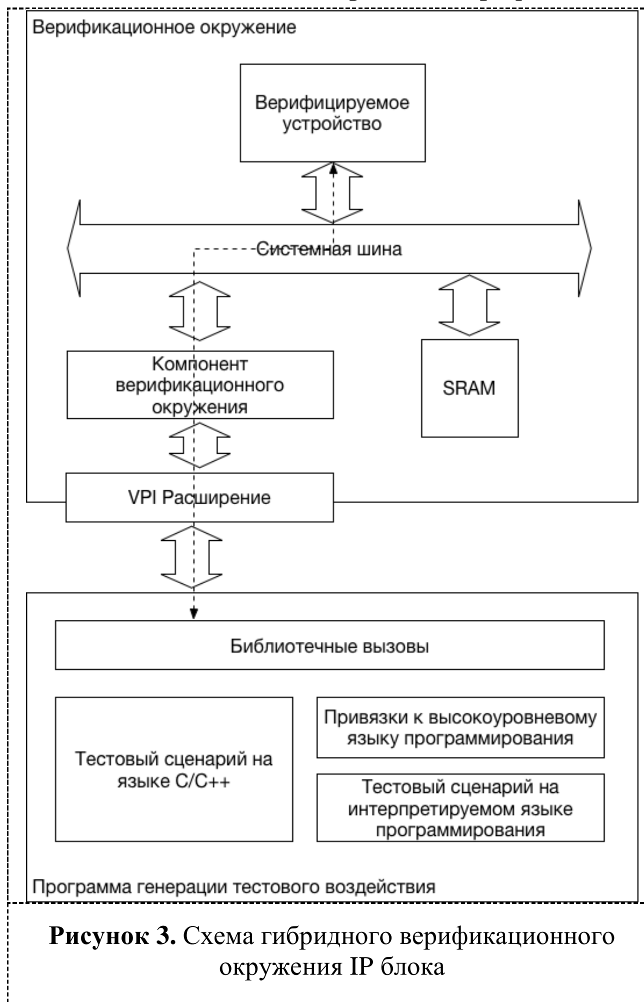

Рисунок 3. Схема гибридного верификационного окружения IP блока

Этот подход[2] основан на наблюдении, что любой тестовый сценарий, задействующий IP блок, сводится к операциям чтения и записи регистров устройства и внутренней памяти.

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. Преимуществом такого подхода является то, что его можно применять даже в верификационном окржуении отдельного IP блока, как показано на Рисунке. 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.

Особенностью данного метода является то, 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.).

Использование контрольных точек моделирования в xcellium

В последних версиях САПР Cadence Xcellium доступна возможность сохранения полного состояния моделирования проекта при помощи вызова специального таска $save(). Состояние сохраняется в базу данных, наравне с моделью СБИС и называется контрольной точкой.

При отсутствии изменений в модели можно полностью восстановить сохраненное состояние моделирования и продолжить с контрольной точки. При восстановлении состояния можно передать дополнительные параметры (plusargs) для модели, которые будут доступны вместе с теми, которые были переданы при начальном запуске. Контрольные точки могут существенно сократить время повторных запусков при наличии в тестах СБИС долгой процедуры инициализации (например, настройка DDR контроллера памяти, PCI-express интерфейса или программная инициализация TLB и т.п.).

Возможность добавления новых параметров (plusargs) для восстанавливаемого состояния модели СБИС позволяет производить дополнительную конфигурацию после восстановления из контрольной точки. (Например, можно загрузку тестового сценария, заданного через plusargs в память). Этот механизм можно использовать для сокращения времени регрессионного тестирования, выполняя “долгую” инициализацию только один единственный раз для всех тестовых сценариев требующих ее. Здесь важно отметить ряд особенностей:

- При использовании механизма plusargs для задания конфигурации модели (например, для выбора подключаемых к модели СБИС блоков), необходимо повторно осуществлять считывание plusargs сразу после вызова таска $save(), а не в блоке initial.
- Невозможно после восстановления из контрольной точки заменить уже исполняемую на моделируемом процессоре программу, так как даже незначительные изменения в коде могут привести к изменению адресов переменных и функций в памяти. Вместо этого необходимо подготавливать две независимые тестовые программы. Одна из них, например, будет исполняться из ROM памяти, и выполнять длительную инициализацию, в то время как вторая часть, содержащая непосредственно тестовый сценарий будет загружаться в накристальное ОЗУ. Контрольную точку при этом следует сохранять после

выполнения процедуры инициализации, непосредственно перед загрузкой и переходом к исполнению программы в накристальном ОЗУ.

Заключение

Оптимизация регрессионного тестирования и непосредственно процесса разработки и тестирования СБИС, это важная задача, которая, как было показано в данной статье, требует учета многих факторов: от тщательного выбора серверного оборудования и его тонкой настройки, заканчивая особенностями проектирования верификационного окружения. Представленные в данной статье способы оптимизации процесса разработки и верификации СБИС не являются единственно возможными. Тем не менее, указанные здесь подходы не требуют ничего сверх базового пакета САПР Cadence и открытых программных инструментов, выполняющих вспомогательную роль, что делает простым их применение.

В статье также не были рассмотрены системы аппаратного ускорения моделирования СБИС, такие как protium [6] из-за объемности данной темы и необходимости дополнительного дорогостоящего оборудования.

Литература

[1] "Использование семейства инструментов CMake для моделирования проектов сложных СБИС в среде Cadence Incisive", Андрианов А.В., Труды НИИСИ РАН, т. 7, № 4, 2017

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

[3] Cadence Incisive Enterprise Simulator Refererence Manual, Cadence 2015

[4] Reducing Snapshot Creation Turnaround for UVM/SV Based TB Using MSIE Approach, Cadence User Conference 2015, https://www.cadence.com/downloads/cdnlive/in/2015/VER2.pdf

[5] CMake: The Cross Platform Build System, Tanner Lovelace, Linux Magazine, Июль 2006 http://clubjuggler.livejournal.com/138364.html

[6] Богданов А.Ю. Опыт применения платформы "Protium" для верификации микропроцессоров. Труды НИИСИ РАН, .– 2017.– том 7, № 2, с. 46-48.