УДК 004.054

РЕАЛИЗАЦИЯ ПОДХОДА STEP-AND-COMPARE ДЛЯ ВЕРИФИКАЦИИ RISC-V ЯДЕР С ИСПОЛЬЗОВАНИЕМ ОТКРЫТОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Карпухин М.С.,

Магистрант,

Национальный исследовательский университет «Московский институт электронной техники»,

г. Москва, г. Зеленоград

Петров Д.В.,

Магистрант,

Национальный исследовательский университет «Московский институт электронной техники»,

г. Москва, г. Зеленоград

Чусов С.А.,

Аспирант,

Национальный исследовательский университет «Московский институт электронной техники»,

г. Москва, г. Зеленоград

Аннотация

Верификация RISC-V процессорных ядер является комплексным процессом, одним из основополагающих этапов которого является выбор подходящих

подходов и техник тестирования. В данной статье рассматриваются основные актуальные подходы и техники к верификации процессорных ядер, основанных на базе открытой архитектуры RISC-V. Анализируются достоинства и недостатки представленных подходов и техник. Основной акцент делается на особенностях реализации техники синхронного сравнения с эталонной моделью (англ. Step-and-Compare). Рассматриваются необходимые, находящиеся в открытом доступе, инструменты, необходимые для использования в рассматриваемой технике.

Ключевые слова: верификация, RISC-V, Step-and-Compare, Spike, симуляция.

IMPLEMENTATION OF THE STEP-AND-COMPARE APPROACH FOR VERIFICATION OF RISC-V CORE USING OPEN-SOURCE SOFTWARE

Karpukhin M.S.,

master's student,

National Research University "Moscow Institute of Electronic Technology",

Moscow, Zelenograd

Petrov D.V.,

master's student,

National Research University "Moscow Institute of Electronic Technology",

Moscow, Zelenograd

Chusov S.A.,

PhD student,

National Research University "Moscow Institute of Electronic Technology",

Moscow, Zelenograd

Ключевые слова: верификация, RISC-V, Step-and-Compare, Spike, симуляция.

Abstract:

Verification of RISC-V processor cores is a complex process, one of the fundamental stages of which is the selection of appropriate testing approaches and techniques. This article discusses the main current approaches and techniques for verifying processor cores based on the open RISC-V architecture. The advantages and disadvantages of the presented approaches and techniques are analysed. The main focus is on the features of implementing the step-and-compare technique. The necessary publicly available tools required for use in the technique under consideration are discussed.

Keywords: verification, RISC-V, Step-and-Compare, Spike, simulation.

Введение

Индустрия микроэлектроники последние несколько десятков лет развивается действительно впечатляющими темпами. В частности, наблюдается рост числа систем на кристалле (СнК), неотъемлемой частью которых являются процессорные ядра. Согласно исследованию компании Siemens 84% проектов заказных интегральных схем включали в себя одно или несколько процессорных ядер. Также в этом исследовании указывается, что число проектов на архитектуре RISC-V составило 58 %, что почти в два раза больше по сравнению с предыдущем исследованием компании [2].

Несмотря на открытость архитектуры RISC-V, на данный момент нет общепризнанной эталонный модели или референсного процессорного ядра, хотя и существуют симуляторы набора команд, но они неидеальны и не покрывают все особое случаи, следовательно требуются исследования в области поиска подходов к верификации. Традиционно в закрытых архитектурах, таких как ARM или x86, эталонная модель предоставляется самим владельцем архитектуры в полностью верифицированном виде, и пользователю остается только провести интеграционные тесты. В случае с открытой архитектурой RISC-V

ответственность за выбор и использование эталонной модели часто ложится на разработчика конкретного ядра [1, 26].

Цель: реализовать подход Step-and-Compare для верификации RISC-V ядер с использованием открытого программного обеспечения.

Задачи:

1. Рассмотреть существующие подходы к верификации процессорных ядер и определить место Step-and-Compare.
2. Провести анализ существующих инструментов для реализации Step-and-Compare подхода.
3. Рассмотреть конкретный вариант реализации подхода Step-and-Compare для верификации RISC-V.

Основная часть

Симуляция и формальная верификация

Для верификации цифровых устройств существует два основных подхода: верификация на основе симуляции и формальная верификация. Рассмотрим подробнее данные подходы.

В ходе верификации на основе симуляции работа HDL-описания устройства моделируется во времени. Моделирование производится при различных тестовых сценариях. Сценарии определяются допустимыми и недопустимыми значениями входных сигналов, а также их последовательностью. Сценарии воздействия на устройство следует из его спецификации. Поведение устройства сравнивается с референсной (эталонной) моделью. Модель, как правило, проектируется при помощи несинтезируемых конструкций языков описания аппаратуры или при помощи высокоуровневых языков программирования. Основная задача в ходе симуляции – сгенерировать

как можно больше таких воздействий, которые приведут к недопустимым состояниям. Стоит заметить, что, чтобы гарантировать отсутствие ошибок в устройстве, необходимо перебрать все возможные комбинации входных сигналов, что в большинстве случаев не является возможным ввиду размеров дизайна, его сложности и жестких временных рамок, которые диктует индустрия.

Второй подход – формальная верификация. Он значительно отличается от верификации на основе симуляции. В ходе формальной верификации описывается набор ограничений на входные и выходные сигналы:

- ограничения на входные сигналы определяют то, какие воздействия могут быть поданы на устройство;
- ограничения на выходные сигналы определяют, каким должен быть результат работы устройства.

Также, опционально, задаются некоторые условия, определяющие в большинстве случаев внутреннее состояние устройства в начале симуляции. В настоящее время описание ограничений осуществляется при помощи подмножества SystemVerilog Assertions (SVA) языка описания аппаратуры SystemVerilog**.** Основное отличие формальной верификации от симуляции состоит в том, что при использовании формальной верификации генерация входных воздействий осуществляется программным обеспечением. Пользователь лишь определяет их диапазон. Далее генерацию и проверку осуществляет специализированное программное обеспечение. Проверка в случае формальной верификации основывается на доказательстве того, что при всех возможных комбинациях входных сигналов с учётом ограничений пользователя устройство окажется только в допустимых состояниях и сгенерирует выходные сигналы, которые удовлетворяют условиям пользователя.

Для верификации RISC-V ядер и процессорных ядер в целом применяется следующая концепция: для верификации целого ядра и большинства его блоков

используется верификация на основе симуляции, а для верификации отдельных, критически важных составляющих, используется формальная верификация. Примеры формальной верификации целых ядер существуют [22], однако их количество значительно меньше, чем проектов на основе симуляции. Работа целого ядра симулируется на различных тестовых программах в сочетании с различными шаблонами генерации внешних воздействий.

Рассматриваемая в статье техника Step-and-Compare относится к верификации на основе симуляции.

Маршрут тестирования RISC-V ядер и место step-and-compare в нём

В большинстве случаев маршрут функциональной верификации RISC-V ядер можно разделить на три основных этапа:

1. Простейшие тесты.
2. Тесты на соответствие спецификации.
3. Продвинутые функциональные тесты.

Простейшие тесты

Целью тестов, относящихся к данному этапу, является быстрое выявление критических, «лежащих на поверхности», ошибок, на начальных этапах тестирования. Они являются минимальным тестовым набором, который обеспечит возможность проведения более комплексных проверок. В таких тестах проверяется сама возможность процессора вычитывать инструкции из памяти и исполнять их. Самыми частыми представителями таких тестов являются “hello-world” тесты и тесты самопроверкой.

Простейшие “hello-world” тесты представляют из себя программу, состоящую из элементарного набора инструкций. Она может состоять, к примеру, всего из пары-тройки вычислительных операций. Выводы о

корректности работы процессорного ядра могут делаться на основе файлов временны́х диаграмм. Схематично этот подход изображён на рисунке 1.

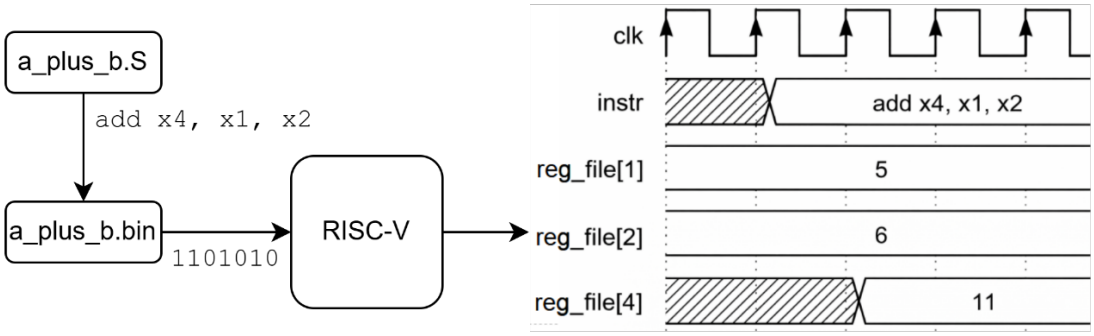

Рис. 1 – Схема подхода “hello-world” тестирования (составлено авторами)

Тесты самопроверкой могут являться следующим этапом после рассмотренного ранее “hello-world” тестирования. В этих тестах объём программы возрастает для того, чтобы увеличить охват проверяемого функционала. Например, эти тесты могут состоять не из пары-тройки, а из я десятков или сотен вычислительных инструкций, результат которых записывается в какую-либо заранее определённую ячейку памяти. Ввиду того, что оценка корректности исполнения программы процессорным ядром по временным диаграммам является крайне времязатратным процессом для такого типа тестируемых устройств, его следует упростить, отчасти переложив этот процесс на само ядро. Этого можно добиться, загружая в какую-либо другую ячейку памяти заранее посчитанное эталонное значение. После исполнения программы можно не искать ошибку на временных диаграммах, а сравнить итоговые значения в двух ячейках. Схематично этот подход изображён на рисунке 2.

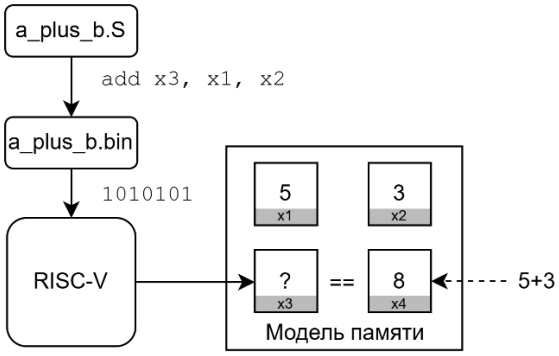

Рис. 2 – Схема теста с самопроверкой (составлено авторами)

Тесты на соответствие спецификации

Поскольку процессорное ядро разрабатывается в соответствии с спецификацией, разработчики должны быть уверены в том, что их продукт действительно выполняет требования, изложенные в соответствующих документах.

В архитектуре RISC-V проводится явное разделением между тестами, направленными на проверку соответствия спецификации (англ. Compliance tests) и тестами, направленными на проверку функциональных особенностей дизайна [9]. Тесты на проверку соответствия спецификации необходимы для обеспечения совместимости с экосистемой программного обеспечения и инструментов. В них проверяется правильно ли разработчик интерпретировал требования ISA (Instruction Set Architecture).

Для проведения такого тестирования существует официальный пакет compliance-тестов – riscv-arch-test [15]. Тесты в этом пакете основаны на сравнении подписей (сигнатур). Сигнатуры – это слепок определённой области памяти, в которую тестовая программа записывает результаты выполнения инструкций. Вместо того, чтобы анализировать каждое действие процессора в реальном времени, метод фокусируется на финальном «отпечатке» его работы в памяти. Этот метод отчасти похож на рассмотренное ранее тестирование самопроверкой с некоторыми важными отличиями. Направленность теста –

проверка соответствия спецификации, а не базовой функциональности, для формирования референсного слепка используется эталонный симулятор набора инструкций, сама сигнатура является более комплексной, нежели просто результат арифметических операций. В сигнатуру, помимо результатов арифметических вычислений могут входить состояния регистров контроля и статуса (CSR), адреса переходов инструкций, результаты выполнения инструкций загрузки и сохранения.

На самом деле тесты на соответствие спецификации являются необходимыми в маршруте верификации RISC-V ядер, но составляют лишь малую часть от общего количества проверок, проводимых над дизайном.

Продвинутые функциональные тесты

В основе данной группы тестов лежит сравнение результатов работы RTL-симулятора и эталонного симулятора архитектуры набора команд.

В этих тестах производится детальный анализ состояния процессорного ядра - анализ исполняемых инструкций и состояний внутренних регистров. Для этой цели можно использовать специальные интерфейсы, контроллер которых на аппаратном уровне встраивается в тестируемую систему. Примерами таких интерфейсов служат: открытый интерфейс RVFI (англ. RISC-V Formal Interface), в настоящее время поддерживаемы компанией YosysHQ [28] и открытый интерфейс RVVI (RISC-V Verification Interface), разрабатываемый компанией Imperas Software совместно с участниками RISC-V сообщества [19]. Подробнее о их различиях будет рассказано в дальнейшем описании методов тестирования.

Эти интерфейсы предоставляют информацию о счётчике команд, выполняемых инструкциях, состоянии регистров общего и специального назначений и т. д.

Для их поддержки процессорное ядро должно содержать специальную логику, реализующую протокол в соответствии со спецификацией на этот интерфейс.

Сравнение файлов трассировки

При подходе сравнения файлов трассировки используется результат (trace log) двух симуляторов: RTL-симулятора и ISA-симулятора.

В ходе симуляции поведения ядра формируется лог-файл его внутреннего состояния, содержащий в основном информацию о содержимом регистров общего и специального назначения, информацию об обращениях в память и содержимом счетчика команд. Для получения информации о состоянии ядра часто используется интерфейс RVFI и специальный несинтезируемый модуль трассировки.

Эталонная модель (независимо от RTL-симулятора) эмулирует работу процессора и так же формирует лог-файл, описывающий состояние ядра в ходе выполнения программы.

Для каждого тестового сценария происходит сравнение лог-файлов процессора и эталонной модели. На основании совпадения или несовпадения формируется результат тестирования.

Поскольку данный подход рассчитан для более комплексной проверки, сложность тестовых программ значительно возрастает. Для их создания в данном подходе используются специальные генераторы случайных инструкций. Схема подхода представлена на рисунке 3.

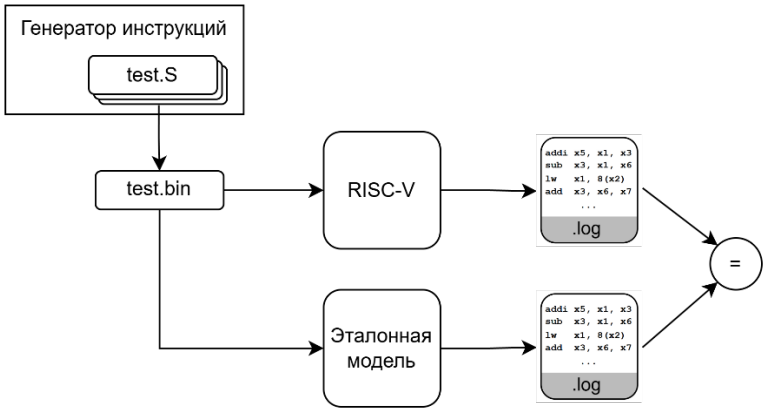

Рис. 3 – Схема подхода сравнения файлов трассировки (составлено авторами)

Подход сравнения файлов трассировки может также использоваться и при тестировании на соответствие спецификации.

В отличие от предыдущих методов тестирования (сравнение сигнатур, hello-world, тесты самопроверкой) этот тест обеспечивает высокое качество проверки благодаря тому, что производится сравнение каждой выполняемой инструкции.

Немаловажным преимуществом (относительно методов, которые будут рассмотрены далее) является простора реализации данного метода.

Основным недостатком являются высокие требования к объёму накопителя для хранения временный файлов симуляции и файлов трассировки.

Кроме того, это приводит к повышенным временны́м затратам, так как симуляция должна дойти до конца, прежде чем будет обнаружен сбой, что замедляет цикл тестирования.

Нельзя не отметить тот факт, что на данный момент нет решений по поддержке проверки асинхронных событий в подходе сравнения файлов трассировки. Теоретически такую возможность реализовать можно, при наличии таких симуляторов и специальных программных средств, которые поддерживают

асинхронные события. На момент написания данной статьи авторы не нашли ни закрытых коммерческих, ни открытых решений.

Синхронное сравнение

Подход синхронного сравнения (англ. Synchronous Step-and-Compare, Sync-Lockstep) представляет собой продвинутый уровень верификации процессоров, при котором эталонная модель и RTL-модель процессора запускаются одновременно в единой среде симуляции. Такой подход в англоязычной литературе называют совместной симуляцией (англ. Co-simulation) [14]. Изменение состояния ядра также отслеживается при помощи интерфейсов RVFI или RVVI.

Как и в случае подхода сравнения файлов трассировки, в синхронном сравнении для создания тестовых программ используются специальные генераторы ассемблерных инструкций, для получения файлов трассировки используется несинтезируемый модуль трассировки, эталонной моделью выступает симулятор набора команд.

В данном подходе при изменении состояния ядра верификационное окружение генерирует запрос об изменении состояния эталонной модели при помощи SystemVerilog DPI-C (механизма, позволяющего осуществлять вызов C/C++ программ непосредственно из SystemVerilog в ходе симуляции). Специальный блок управления в верификационном окружении, как бы, «шагает» обеими моделями, извлекает данные из RTL через трассировщик и запрашивает аналогичные данные из эталонной модели. Информация попадает в окружение, и производится сравнение состояния тестируемого ядра и референсной модели. В большинстве случаев опрос модели происходит каждый раз, когда тестируемое ядро сигнализирует о выполнении инструкции в ходе симуляции. Схема подхода представлена на рисунке 4

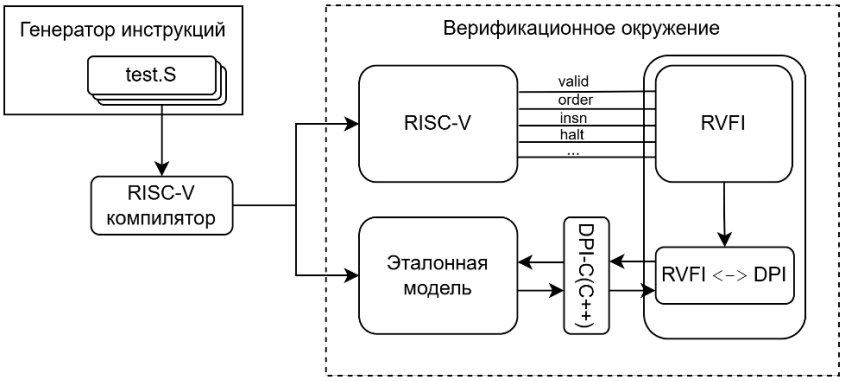

Рис. 4 – Схема подхода Step-and-Compare (составлено авторами)

Проверке подлежат счётчик команд, значения в регистрах общего назначения, состояние регистров контроля и статуса и другие внутренние состояния процессора.

Поскольку данном подходе сравнение поведения ядра и его эталонной модели производится в ходе симуляции, это решает проблему нехватки дискового пространства при генерации большого объема временных файлов и инкапсулирует процесс проверки в процесс симуляции, делая его более явным. Поскольку тест останавливается сразу в момент расхождения состояния моделей, это значительно ускоряет процесс верификации, так как тестировщику не приходится ждать окончания симуляции и искать место ошибки анализируя длинный log-файл симуляции. Однако синхронное сравнение также не поддерживает верификацию ядра с учетом асинхронных событий в его базовой реализации. Из-за различий в микроархитектуре RTL-модель и эталонная модель могут принимать прерывания в разные моменты времени, что приводит к ложному расхождению состояний.

Немаловажным аспектом является сложность реализации данного подхода.

Асинхронное сравнение

Часть недостатков предыдущего подхода решает подход асинхронного сравнения (англ. Asynchronous Lockstep Compare, Asynchronous Step-and-

Compare). На момент написания статьи этот подход является наиболее совершенным с точки зрения качества верификации и обнаружения ошибок. Он является логическим развитием синхронного сравнения, добавляя возможность обработки прерываний, запросов отладки и других асинхронных сигналов, точное время возникновения которых в реальном конвейере и абстрактной модели может различаться. Схема подхода представлена на рисунке 5.

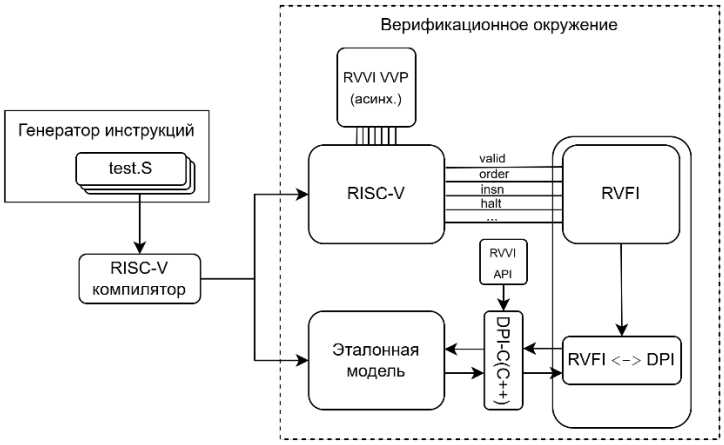

Рис. 5 – Схема подхода асинхронного Step-and-Compare (составлено авторами)

В основе этого подхода лежит интеграция RTL-модели и эталонного симулятора в едину среду совместной симуляции, как и в предыдущем подходе, за исключением того, что в этом подходе добавлен компонент анализа асинхронных событий. Когда RTL-модель реагирует на событие (например, переходит на вектор прерывания), компонент анализирует текущее состояние и определяет, является ли такая реакция допустимой согласно спецификации ISA в данный момент времени. Если реакция признана корректной, эталонная модель принудительно переводится в соответствующее состояние, что позволяет продолжать сравнение без ложных срабатываний из-за расхождения во времени прихода прерывания.

Недостатком этого метода является крайне высокая сложность реализации.

Обоснование выбора применяемых инструментов

На данный момент на рынке существует не так много решений, реализующих подход step-and-compare. Одним из самых популярных, является решение от компании Synopsys – ImperasDV. В нём реализован комплексный подход к верификации RISC-V ядер [23]. Присутствует поддержка как синхронного, так и асинхронного пошагового сравнения. В качестве эталонной модели выступает их собственная модель – ImperasFPM RISC-V Model, симуляция RTL происходит при помощи симулятора Synopsys VCS [24], интерфейс – RVVI, а для отладки и анализа используется Synopsys Verdi Debug System [25]. Это решение действительно является комплексным, всеобъемлющим, позволяющим значительно увеличить скорость и качество верификации. Однако, основным недостатком является закрытый исходный код.

В открытом сообществе также не существует реализаций, аналогичным ImperasDV, поэтому у производителей оборудования остаётся небольшой выбор: либо использовать проприетарное решение от Synopsys, либо собирать своё окружение на базе открытых инструментов. На момент написания статьи не представляется возможным использовать подход асинхронного Step-and-Compare, на базе открытых инструментов, так как возможности открытых инструментов сильно ограничены, поэтому в данной статье реализуется синхронный Step-and-Compare.

На данный момент, в сообществе существуют следующие реализации основных инструментов:

Генераторы инструкций: RISCV-DV [4], MicroTESK [7], AAPG [20], FORCE-RISCV [12].

В настоящее время самым популярным генератором случайных RISC-V инструкций является RISCV-DV. Однако его использование возможно только при наличии доступа к коммерческим симуляторам, таким как Siemens QuestaSim,

Synopsys VCS, Cadence Xcelium. В качестве аналога можно использовать генератор инструкций AAPG написанный на Python.

RTL-симуляторы: Siemens QuestaSim [20], Synopsys VCS, Cadence Xcelium [3], Verilator [27], Icarus Verilog [13].

Из ранее перечисленных открытым являются Icarus Verilog и Verilator. Оба они трудно адаптируемы для больших проектов, так как не поддерживают множество функций и концепций, которые поддерживают коммерческие симуляторы (например, UVM [6]). Первый является классическим симулятором, который передвигается в симуляции по определённым событиям (event-based). Verillator же, теоретически, должен работать быстрее, так как он работает с кодом, скомпилированным из Verilog в C++. Немаловажным является тот факт, что Verillator интенсивно развивается благодаря большой поддержке сообщества, поэтому выбор пал на него.

ISA-симуляторы: riscvOPVsim [16], Spike [17], VeeR-ISS [5]

Выбор пал на симулятор Spike ввиду того, что он пользуется огромной поддержкой со стороны сообщества.

Верификационные окружения: Ibex [8], CORE-V Verif [11], ImperasDV.

Представленные окружения рассчитаны на работу с RISCV-DV генератором инструкций (кроме ImperasDV, он не заточен под конкретный генератор инструкций). Поэтому был выбран подход с написанием своего собственного верификационного окружения.

Интерфейс RVVI является логическим развитием интерфейса RVFI и используется, в основном, в асинхронном Step-and-Compare. Поскольку было принято решение реализовывать синхронное сравнение, поставленных задач можно добиться использованием RVFI.

Маршрут реализации верификационного окружения с использованием step-and-compare

Для реализации верификационной техники Step-and-Compare используется следующие окружение (рис.6).

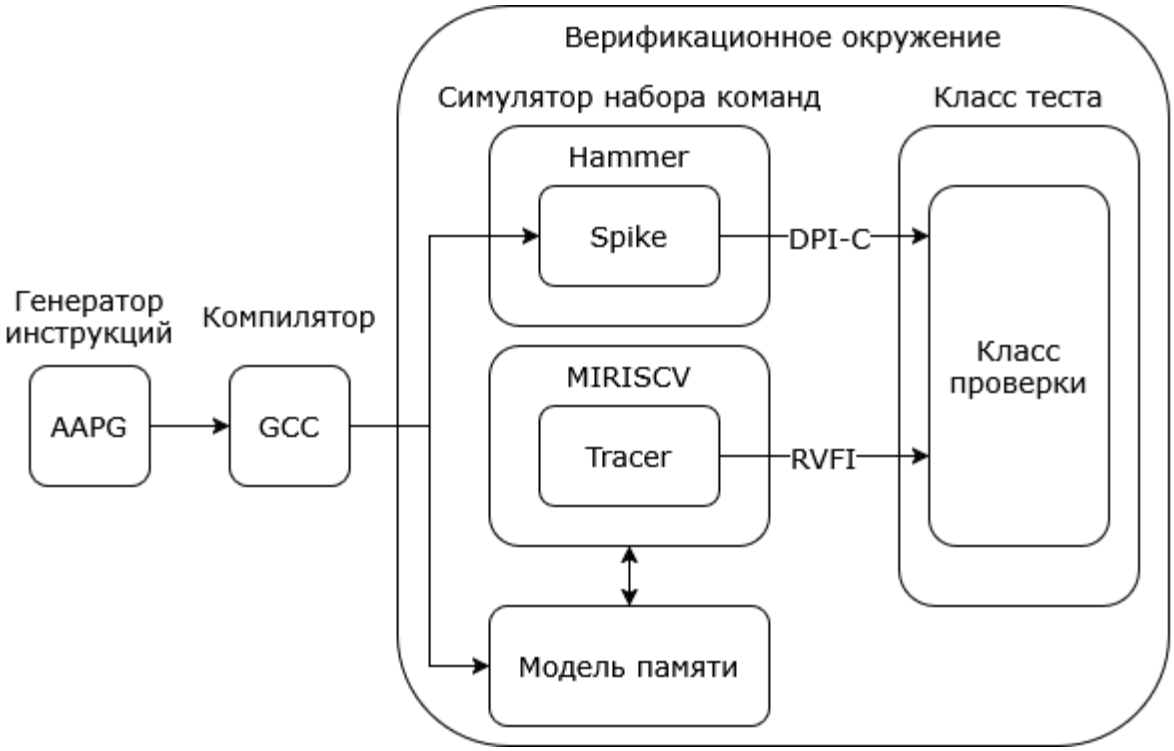

Рис. 6 – Схема верификационного окружения (составлено авторами)

Для тестирования используется учебное ядро, разработанное НИЛ ЭСК НИУ МИЭТ: MIRISCV [10]. Процессор поддерживает расширения RV32I и RV32M и является четырехстадийным in-order конвейером. Ядро поддерживает только машинный уровень привилегий.

Подробнее рассмотрим механизм работы верификационного окружения [18]. Генератор инструкций создает потоки ассемблерных инструкций, которые после компиляции одновременно поступают в виде машинного кода на симулятор набора команд и на тестируемое ядро. Для работы программного кода на MIRISCV требуется модель памяти, в которой по соответствующим адресам размещается тестовая программа. В процессорное ядро интегрирован аппаратный блок трассировки (tracer) для мониторинга и записи истории

выполнения инструкций. Для вывода информации об исполнении команд используется RVFI (RISC-V Formal Interface) — это стандартизированный интерфейс для формальной верификации процессорных ядер RISC-V. Он представляет собой набор выходных сигналов, которые предоставляют полную трассу выполнения инструкций, включая метаданные, операции с регистрами и памятью, а также поток управления. Для взаимодействия с эталонной моделью используется библиотека Hammer, которая предоставляет набор высокоуровневых методов (Application Level Interface, API) для работы с Spike, которые являются методами SystemVerilog DPI-C. Информация о состоянии модели попадает в окружение, и производится синхронное сравнение с состоянием тестируемого процессорного ядра. Остальные компоненты верификационного окружения описаны на SystemVerilog с использованием концепции ООП. Для сравнения результатов выполнения инструкций используется класс проверки. Для взаимодействия эталонной модели и тестируемого ядра с механизмом сравнения окружение используется класс теста, который является связующим звеном для интерпретации RVFI и DPI-C.

Подробнее рассмотрим использование Hammer. Ниже представлена таблица с кратким описанием каждого DPI-C метода (таблица 1).

Таблица 1 – Методы Hammer

Название	Описание
hammer_init()	Создание объекта типа Hammer и возвращение указателя на него.
hammer_release()	Удаление объекта типа Hammer.
hammer_single_step()	Выполнение одной инструкции.
hammer_get_PC()	Получение текущего значения счетчика команд.
hammer_set_PC()	Установка значения счетчика команд.

hammer_get_insn_str()	Получение строкового ассемблерного представления инструкции, которая находится в памяти по адресу текущего счетчика команд.
hammer_get_gpr()	Получение текущего значения регистра общего назначения.

В оригинальной библиотеке отсутствовал метод для получения инструкции, которая выполняется Spike на текущем счетчике команд. Без этого невозможно реализовать Step-and-Compare, поэтому он был добавлен. Реализация метода представлена на рисунке 7.

```
uint64_t Hammer::get_insn_bits(uint8_t hart_id) {
    processor_t *hart = simulator->get_core(hart_id);
    mmu_t *mmu = hart->get_mmu();
    reg_t pc = hart->get_state()->pc;
    insn_fetch_t fetch = mmu->load_insn(pc);
    insn_t insn = fetch.insn;
    return insn.bits();
}
```

Рис. 7 – Реализация метод для получения инструкции, которая выполняется на текущем счетчике команд (составлено авторами)

Метод принимает идентификатор ядра hart_id в качестве аргумента (по образу и подобию иных методов, объявленных в файле) и возвращает 64-битное беззнаковое целое число uint64_t

Строка processor_t *hart = simulator->get_core(hart_id) необходима для получения указателя на ядро процессора с идентификатором hard_id. Так как Spike может эмулировать поведение многоядерного процессора, доступ к конкретному ядру осуществляется при помощи метода simulator->get_core().

simulator – указатель на Spike, который создается выше в рассматриваемом файле. Тип processor_t подробно разбираться не будет.

Строка mmu_t *mmu = hart->get_mmu() необходима для получения указателя на модуль управления памятью (Memory Management Unit, MMU) конкретного ядра. Тип mmu_t подробно разбираться не будет.

Строка reg_t pc = hart->get_state()->pc необходима для получения текущего счетчика команд ядра. Тип reg_t – это тип uint64_t, как было определено ранее.

Строка insn_fetch_t fetch = mmu->load_insn(pc) необходима для получения структуры текущей инструкции fetch при помощи метода mmu->load_insn(). Инструкция загружается по адресу, который определяется счетчиком команд pc.

Для взаимодействия со Spike в библиотеке Hammer создается класс типа Hammer. Метод hammer_init() спроектирован таким образом, что возвращает указатель на область памяти, где будет располагаться объект созданного класса. Указатель возвращается через тип SystemVerilog chandle. SystemVerilog окружение далее может обращаться к объекту типа Hammer при помощи этого указателя.

Этот указатель передается в остальные методы в качестве аргумента с названием hammer. Метод hammer_release() создан для уничтожения объекта типа Hammer. Другие методы – для взаимодействия с объектом типа Hammer с целью получения состояния Spike в текущий момент времени симуляции, а также изменения его состояния через выполнение инструкции (hammer_single_step()).

Также заметим, что методы для получения и установки значений регистров возвращают и принимают значения типа bit[31:0], так как в тестируемом ядре счетчик команд и регистры общего назначения 32-битные. Также тестируемое ядро содержит 32 регистра общего назначения, вследствие чего тип аргумента gpr_id метода hammer_get_gpr() имеет ширину 5 бит.

Далее рассмотрим обработку информации с RVFI и сравнения с эталонной моделью. Вся обработка реализована внутри класса теста, который отвечает за различные функциональные настройки и симуляцию. Можно выделить основные задачи класса:

- получение адреса условия завершения тестирования;
- инициализацию памяти;
- запуск основной задачи класса агента;
- запуск основной задачи класса последовательности;
- ожидание завершения тестирования по истечении времени;
- ожидание завершения тестирования по условию.

Подробная структура представлена на рисунке 8.

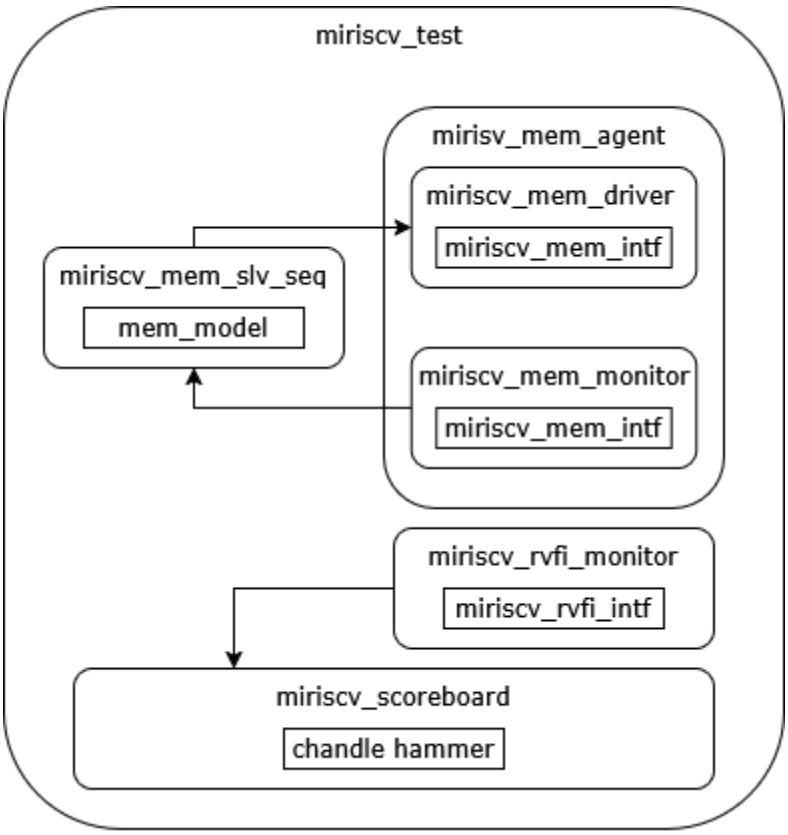

Рис. 8 – Структура класса теста (составлено авторами)

В окружении для взаимодействия процессора с памятью используется подход с обратной связью. Класс miriscv_mem_monitor отслеживает запросы

ядра в память и отправляет информацию в класс последовательности miriscv_mem_seq, в которой находится указатель (object handle) на универсальную модель памяти mem_model. Последовательность в свою очередь формирует транзакции miriscv_mem_item для класса miriscv_mem_driver, который отвечает за взаимодействие окружения с сигналами процессорного ядра. Классы miriscv_mem_monitor и miriscv_mem_driver инкапсулированы в класс miriscv_mem_agent, который в свою очередь находится в классе miriscv_test, который создает и инициализирует модель памяти, определяет тестовый сценарий.

Для получения информации о выполненных инструкциях отвечает класс монитор miriscv_rvfi_monitor, который будет отслеживать сигналы RVFI, генерируемые процессорным ядром, и отправлять текущее состояние этих сигналов в класс проверки miriscv_scoreboard. Подробнее рассмотрим пример реализации класса проверки (рис. 9).

```
class miriscv_scoreboard;
  mailbox#(miriscv_rvfi_item) rvfi_mbx;
  chandle hammer;
  protected int unsigned retire_cnt;
  protected miriscv_insn_info_s cur_insn_info;

  virtual function void init(string elf, bit [31:0] pc);
    hammer = hammer_init(elf);
    hammer_set_PC(hammer, pc);
    $display("Hammer was initialized with PC: %8h", pc);
  endfunction

  virtual function int unsigned get_retire_cnt(); return retire_cnt;
  endfunction

  virtual function miriscv_insn_info_s get_cur_insn_info(); return cur_insn_info;
  endfunction

  virtual task run();
    miriscv_rvfi_item t;
    forever begin
      rvfi_mbx.get(t);
      void'(check_pc_and_instr(t));
      hammer_single_step(hammer);
      void'(check_rd(t));
      retire_cnt = retire_cnt + 1;
    end
  endtask

  virtual function bit check_pc_and_instr(miriscv_rvfi_item t);
    bit result = 1; string msg;
    cur_insn_info.pc   = hammer_get_PC       (hammer);
    cur_insn_info.bits = hammer_get_insn_bits(hammer);
    cur_insn_info.str  = hammer_get_insn_str (hammer);
    if( cur_insn_info.pc !== t.rvfi_pc_rdata ) begin
      msg = "\nPC mismatch! "; result = 0;
    end
    if( cur_insn_info.bits !== t.rvfi_insn ) begin
      msg = {msg, "Instruction mismatch!"}; result = 0;
    end
    if( !result ) begin
      msg = {msg, $sformatf("\nHammer PC: %8h insn: %8h (%s) \nMIRISCV PC: %8h insn: %8h",
        cur_insn_info.pc, cur_insn_info.bits, cur_insn_info.str, t.rvfi_pc_rdata, t.rvfi_insn)};
      $display(msg);
    end
    return result;
  endfunction

  virtual function bit check_rd(miriscv_rvfi_item t);
    bit result = 1; string msg;
    cur_insn_info.rd = hammer_get_gpr(hammer, t.rvfi_rd_addr);
    if( cur_insn_info.rd !== t.rvfi_rd_wdata ) begin
      msg = "\nRD mismatch! ";
      msg = {msg, $sformatf("PC: %8h insn: %8h (%s)",
          cur_insn_info.pc, cur_insn_info.bits, cur_insn_info.str)};
      result = 0;
    end
    if( !result ) begin
      msg = {msg, $sformatf("\nHammer  x%0d: %8h \nMIRISCV x%0d: %8h",
          t.rvfi_rd_addr, cur_insn_info.rd, t.rvfi_rd_addr, t.rvfi_rd_wdata)};
      $display(msg);
    end
    return result;
  endfunction
endclass
```

Рис. 9 – Пример реализации класса проверки (составлено авторами)

Класс содержит параметризованный mailbox с именем rvfi_mbx для получения информации от монитора, а также указатель hammer на объект типа Hammer, который будет использован для взаимодействия с соответствующей библиотекой. Также класс содержит защищенный счетчик количества проверенных инструкций retire_cnt и информацию о текущей проверенной инструкции cur_insn_info.

В методе init() следствием вызова DPI-C метода hammer_init() является то, что переменная hammer начинает указывать на место в памяти с объектом из библиотеки Hammer, при помощи которого можно взаимодействовать со Spike. Также передается путь до исполняемого файла через аргумент. Далее устанавливается начальное значение счетчика команд при помощи вызова DPI-C метода hammer_set_PC().

В методе run() после получения информации от RVFI вызывается метод проверки счетчика команд и выполненной инструкции при помощи check_pc_and_instr(). Вызов метода обернут в приведение к void, т.к. помимо вывода информации возвращает статус проверки (0 или 1). В качестве аргумента в метод передается транзакция, содержащая информацию о состоянии тестируемого процессорного ядра. Далее производится вызов DPI-C метода hammer_single_step() для выполнения одной инструкции на Spike при помощи Hammer. Далее производится сравнение значения в регистре назначения в процессорном ядре и в Spike при помощи check_rd(). Вызов метода check_rd() так же, как и метода check_pc_and_instr(), обернут в приведение к void и принимает такой же аргумент.

В методе check_pc_and_instr() производится сравнение счетчика команд тестируемого процессорного ядра MIRISCV и счетчика команд в Spike. Значение счетчика команд MIRISCV, по адресу которого находилась выполненная

инструкция, содержится в сигнале rvfi_pc_rdata. Значение счетчика команд Spike определяется при помощи вызова DPI-C метода hammer_get_PC().

В методе check_rd() производится сравнение регистра назначения тестируемого процессорного ядра MIRISCV и регистра назначения в Spike. Под регистром назначения подразумевается регистр, куда записывается результат выполнения инструкции. Значение регистра назначения конкретной инструкции в случае MIRISCV содержится в сигнале rvfi_rd_wdata . Значение регистра назначения Spike определяется при помощи вызова DPI-C метода hammer_get_gpr.

В обоих методах информация об инструкции, полученная при помощи DPI-C методов, хранится в структуре cur_insn_info.

Заключение

Верификация RISC-V процессорных ядер требует комплексного подхода с тщательным выбором методов тестирования. Среди современных техник особое внимание уделяется синхронному сравнению с эталонной моделью (Step-and-Compare), которое позволяет проверять корректность работы ядра относительно стандартной модели поведения. Для реализации этой техники эффективными являются открытые инструменты, такие как Spike, при этом использование библиотеки Hammer облегчает интеграцию эталонного исполнения с тестовым окружением, написанным на SystemVerilog. Таким образом, сочетание открытых моделей и Step-and-Compare обеспечивает надёжную и прозрачную верификацию RISC-V процессорных ядер, позволяя выявлять ошибки на ранних стадиях разработки.

Библиографический список

1. Чусов С. А. Особенности современных подходов к верификации RISC-V-ядер // Наноиндустрия. — 2024. — Т. 17. — № S10-2 (128). — С. 768–776. — ISSN 1993-8578.

2. 2024 Wilson Research Group IC/ASIC functional verification trend report // Siemens Software Resources. — URL: https://resources.sw.siemens.com/en-US/white-paper-2024-wilson-research-group-ic-asic-functional-verification-trend-report/ (Дата обращения: 24.12.2025).

3. Cadence Design Systems, Inc. Xcelium Simulator: High□Performance Simulator for Functional Verification / Cadence Design Systems, Inc. — *Cadence*. — [Электронный ресурс]. — Режим доступа — URL: https://www.cadence.com/en_US/home/tools/system□design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html (Дата обращения: 24.12.2025).

4. Chipsalliance. RISC-DV / chipsalliance // GitHub. — [Электронный ресурс]. — Режим доступа — URL: https://github.com/chipsalliance/riscv□dv (Дата обращения: 24.12.2025).

5. Chipsalliance. VeeR□ISS / chipsalliance // GitHub. — [Электронный ресурс]. — Режим доступа — URL: https://github.com/chipsalliance/VeeR□ISS (Дата обращения: 24.12.2025).

6. IEEE Std 1800.2□2020. IEEE Standard for Universal Verification Methodology Language Reference Manual / IEEE Computer Society. — *IEEE Standards*. — [Электронный ресурс]. — Режим доступа — URL: https://standards.ieee.org/ieee/1800.2/7567/ (Дата обращения: 24.12.2025).

7. ISP RAS. MicroTESK□RISC□V / ISP RAS // *Forge.ISPRAS.ru*. — [Электронный ресурс]. — Режим доступа — URL: https://forge.ispras.ru/projects/microtesk□riscv (Дата обращения: 24.12.2025).

8. LowRISC. Ibex: Open□Source RISC□V Processor / lowRISC // GitHub. — [Электронный ресурс]. — Режим доступа — URL: https://github.com/lowRISC/ibex (Дата обращения: 24.12.2025).

9. McDermott K. Getting Started with RISC-V Verification / K. McDermott // *RISC-V International Blog*. — 2020. — [Электронный ресурс]. — Режим доступа — URL: https://riscv.org/blog/getting-started-with-risc-v-verification/ (Дата обращения: 24.12.2025).

10. MPSU. MIRISCV / MPSU // GitHub. — [Электронный ресурс]. — Режим доступа — URL: https://github.com/MPSU/MIRISCV (Дата обращения: 24.12.2025).

11. OpenHW Group. core□v□verif / OpenHW Group // GitHub. — [Электронный ресурс]. — Режим доступа — URL: https://github.com/openhwgroup/core□v□verif (Дата обращения: 24.12.2025).

12. OpenHW Group. force□riscv / OpenHW Group // GitHub. — [Электронный ресурс]. — Режим доступа — URL: https://github.com/openhwgroup/force□riscv (Дата обращения: 24.12.2025).

13. Pablo Bleyer Kocik. Icarus Verilog for Windows / Pablo Bleyer Kocik // *bleyer.org*. — [Электронный ресурс]. — Режим доступа — URL: https://bleyer.org/icarus/ (Дата обращения: 24.12.2025).

14. Qiu R., Liu Y. An Integrated UVM-TLM Co-Simulation Framework for RISC-V Functional Verification and Performance Evaluation / R. Qiu, Y. Liu. — 2025. — [Электронный ресурс]. — Режим доступа — URL: https://arxiv.org/abs/2505.10145 (Дата обращения: 24.12.2025).

15. RISC-V Architecture Test SIG. riscv-arch-test / riscv-non-isa // GitHub. — [Электронный ресурс]. — Режим доступа — URL: https://github.com/riscv-non-isa/riscv-arch-test (Дата обращения: 24.12.2025).

16. Riscv-admin. riscv□ovpsim / riscv□admin // GitHub. — [Электронный ресурс]. — Режим доступа — URL: https://github.com/riscv□admin/riscv□ovpsim (Дата обращения: 24.12.2025).

17. Riscv-software-src. riscv□isa□sim / riscv□software□src // GitHub. — [Электронный ресурс]. — Режим доступа — URL: https://github.com/riscv□software□src/riscv□isa□sim (Дата обращения: 24.12.2025).

18. Riscv-tests-intro. riscv□tests□intro / riscv□tests□intro // GitHub. — [Электронный ресурс]. — Режим доступа — URL: https://github.com/riscv□tests□intro/riscv□tests-intro (Дата обращения: 24.12.2025).

19. RVVI. RVVI / riscv-verification // GitHub. — [Электронный ресурс]. — Режим доступа — URL: https://github.com/riscv-verification/RVVI (Дата обращения: 24.12.2025).

20. SHAKTI Project. AAPG: Automated Assembly Program Generator for the RISC□V ISA / SHAKTI Project // *GitLab*. — [Электронный ресурс]. — Режим доступа — URL: https://gitlab.com/shaktiproject/tools/aapg (Дата обращения: 24.12.2025).

21. Siemens Digital Industries Software. Questa One: smart verification solution / Siemens Digital Industries Software // *Siemens EDA*. — [Электронный ресурс]. — Режим доступа — URL: https://eda.sw.siemens.com/en-US/ic/questa-one/ (Дата обращения: 24.12.2025).

22. SymbioticEDA. RISC-V Formal Verification Framework / SymbioticEDA // GitHub. — [Электронный ресурс]. — Режим доступа — URL: https://github.com/SymbioticEDA/riscv-formal (Дата обращения: 24.12.2025).

23. Synopsys Inc. ImperasDV: RISC-V Processor Verification Solution / Synopsys Inc. — *Synopsys*. — [Электронный ресурс]. — Режим доступа — URL: https://www.synopsys.com/verification/imperasdv.html (Дата обращения: 24.12.2025).

24. Synopsys Inc. VCS: Functional Verification Solution / Synopsys Inc. // *Synopsys*. — [Электронный ресурс]. — Режим доступа — URL:

https://www.synopsys.com/verification/simulation/vcs.html (Дата обращения: 24.12.2025).

25. Synopsys Inc. Verdi: Debug and Verification Management Platform / Synopsys Inc. — *Synopsys*. — [Электронный ресурс]. — Режим доступа — URL: https://www.synopsys.com/verification/debug/verdi.html (Дата обращения: 24.12.2025).

26. Tain C., Patil S., Al-Asaad H. Survey of Verification of RISC-V Processors // *Journal of Electronic Testing: Theory and Applications*. 2025. Vol. 41, pp. 111–138. — URL: https://link.springer.com/article/10.1007/s10836-025-06169-3 (Дата обращения: 24.12.2025).

27. Veripool. Verilator: Fast Open□Source Verilog/SystemVerilog Simulator / Veripool // *Veripool.org*. — [Электронный ресурс]. — Режим доступа — URL: https://www.veripool.org/verilator/ (Дата обращения: 24.12.2025).

28. YosysHQ. riscv-formal / YosysHQ // GitHub. — [Электронный ресурс]. — Режим доступа — URL: https://github.com/YosysHQ/riscv-formal (Дата обращения: 24.12.2025).