MALT system

Руководство пользователя

Вентилятор

ПАК для ускорения верификации RTL

Руководство по использованию ПАК для ускорения верификации RTL "Вентилятор"

Выпуск 0.5

MALT

апр. 16, 2026

Оглавление:

ГЛАВА 1

Введение

ПАК «Вентилятор» (далее «Вентилятор») - это программно-аппаратное средство предназначенное для ускорения симуляции и верификации RTL кода. «Вентилятор» позволяет ускорить верификацию СнК и других RTL проектов за счёт совмещения преимуществ поведенческих симуляторов и прототипирования на ПЛИС. С одной стороны засчёт аппаратного ускорения позволяет моделировать RTL существенно быстрее чем классические поведенческие симуляторы, а с другой «Вентилятор» предоставляет более удобную интеграцию HDL кода с моделями, более быстрый старт симуляции и более простое получение отладочной информации чем в случае макетирования на ПЛИС.

Настоящее руководство содержит информацию о работе с «Вентилятор» с точки зрения пользователя. Руководство рассчитано на пользователей знакомых с симуляцией RTL в классических поведенческих HDL-симуляторах (Siemens QuestaSim, Cadence Xcelium, Icarus Verilog и др.) и желающих ускорить верификацию своего RTL кода при помощи «Вентилятор». Предполагается что пользователю знакомы основные понятия из области разработки и верификации RTL кода, а также основные принципы работы в терминале Linux.

ГЛАВА 2

Руководство по быстрому началу работы

2.1 Запуск симуляции примера

Для быстрого ознакомления с возможностями и начала работы с «Вентилятор» рекомендуется запустить на моделирование один из примеров идущих в комплекте с ПО «Вентилятор». Для этого:

1. Зайти (локально или через удаленный рабочий стол) на хост-компьютер «Вентилятор».
2. В терминале перейти в папку с примерами по адресу *$HOME/examples/$ИМЯ_ПРИМЕРА* , где $ИМЯ_ПРИМЕРА - название папки любого из приложенных примеров (см. *Приложение: Примеры дизайнов для симуляции*), например neorv32.
3. Запустить проект на симуляцию при помощи команды:

```
ventilator_run $ИМЯ_ПРИМЕРА.json
```

4. Дождаться завершения симуляции сопровождающегося сообщением *Verilator flow completed succesfully!*.
5. Изучить полученный консольный вывод процесса симуляции (в терминале) и вейвформу в открывшемся окне.

2.2 Запуск симуляции своего проекта

Для первого запуска симуляции своего примера необходимо:

1. Зайти (локально или через удаленный рабочий стол) на хост-компьютер «Вентилятор».
2. Перенести исходные файлы проекта для симуляции на хост-компьютер «Вентилятор».
3. Создать конфигурационный файл проекта в формате JSON. Для упрощения этой задачи можно скопировать конфигурационный файл JSON одного из примеров и изменить в нём названия файлов (SOURCES) и топ-модуля (TOPMOD) в соответствии со своим проектом. Подробнее о содержимом конфигурационного файла см. *Конфигурационный файл проекта*.
4. Запустить проект на симуляцию при помощи команды:

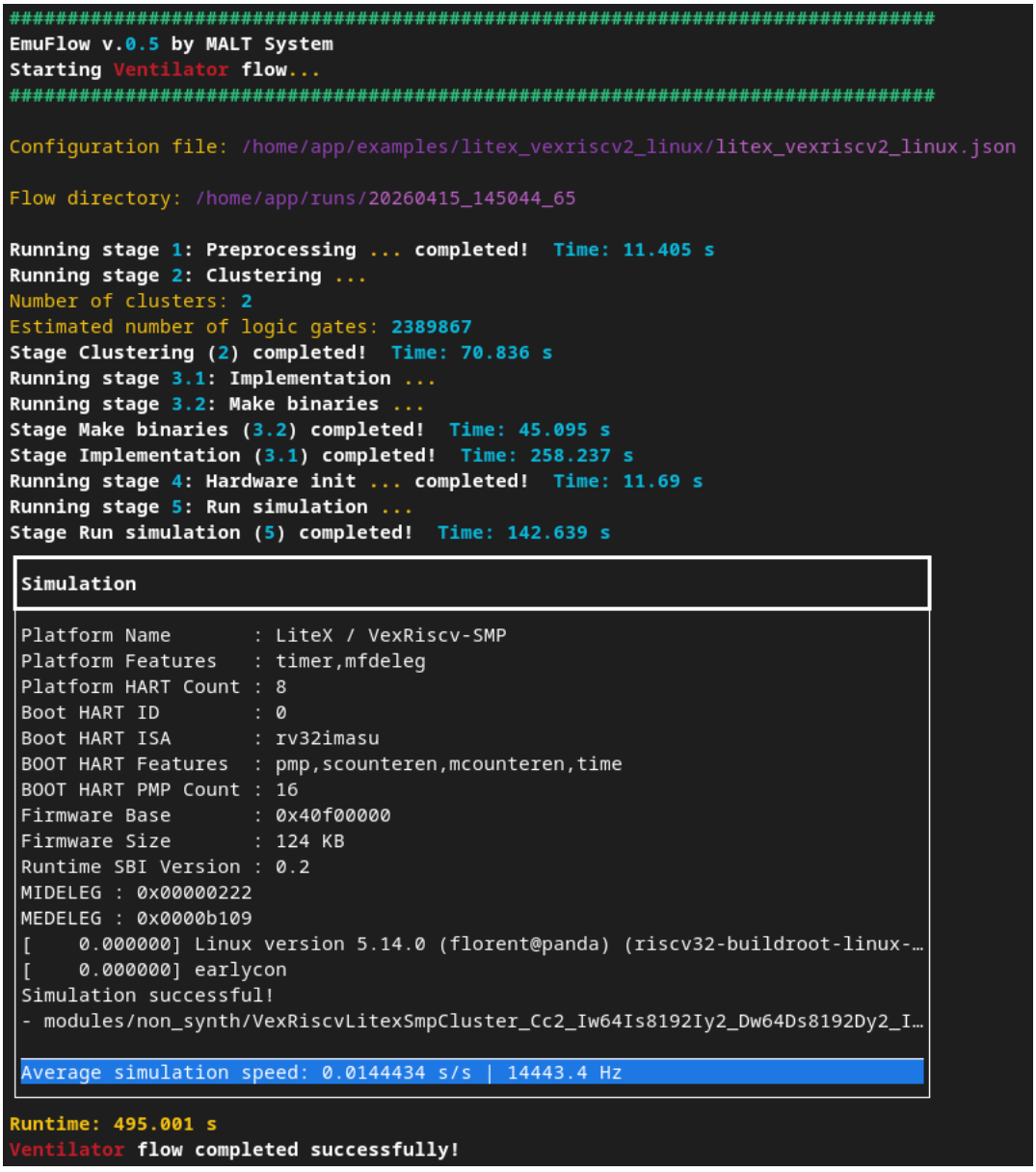

Рис. 1: Пример терминального вывода успешно завершенной симуляции на «Вентилятор»

```
ventilator_run $КОНФИГ_ФАЙЛ.json
```

5. Дождаться завершения симуляции сопровождающегося сообщением *Verilator flow completed succesfully!*.
6. Изучить полученный консольный вывод процесса симуляции (в терминале) и вейвформу в открывшемся окне.

ГЛАВА 3

Состав «Вентилятор»

3.1 Аппаратные компоненты

Аппаратная часть «Вентилятор» состоит из аппаратного модуля ускорения симуляции и хост-компьютера под управлением ОС Linux. Аппаратный модуль ускорения содержит в своём составе массив микросхем ПЛИС на которые отображается синтезируемая часть симулируемого RTL-проекта.

Всё взаимодействие пользователя с «Вентилятор» производится с хост-компьютера, соответственно все действия настоящего руководства подразумевают, что пользователь работает на хост-компьютере (локально или удалённо, например через VNC). При этом всё необходимое ПО уже установлено и настроено на хост-компьютере на момент поставки, пользователю необходимо лишь на него залогиниться.

3.2 Программные компоненты

Программные компоненты «Венилятор» включают в себя ПО для симуляции, ПО обеспечивающее интерфейс с пользователем и набор дизайнов-примеров для симуляции.

ПО для симуляции осуществляет функционирование комплекса, пользователь напрямую с ним не взаимодействует. Общая схема работы этого ПО представлена на Рис. 2. ПО основывается на open source инструментах для работы с HDL, в частности в качестве симулятора для исполнения несинтезируемой части используется симулятор Verilator, а синтез на ПЛИС для аппаратного ускорения производится при помощи синтезатора Yosys.

ПО обеспечивающее интерфейс с пользователем подробно описывается в разделе *Интерфейс симуляции*.

Примеры располагаются в директории *$HOME/ventilator/examples/* и предназначены для ознакомления пользователя с возможностями Вентилятор и особенностями реализации конфигурационных файлов для различных ситуаций. Некоторые примеры описаны в разделе *Приложение: Примеры дизайнов для симуляции*.

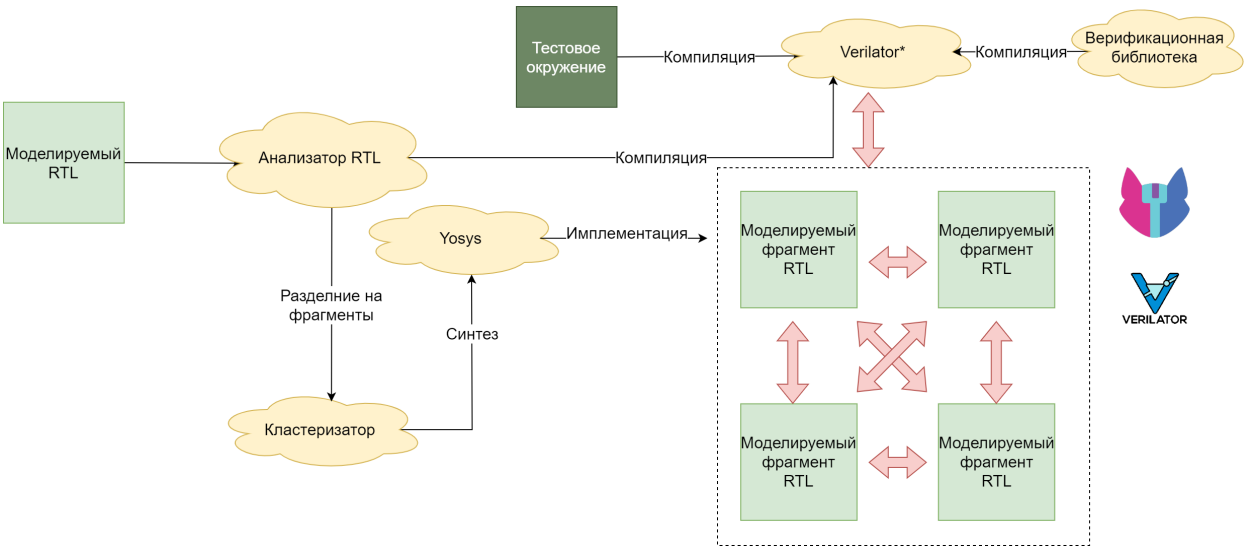

Рис. 2: Схема работы ПО для симуляции

ГЛАВА 4

Возможности и ограничения

4.1 Синтезируемое и несинтезируемое подмножество

Поскольку RTL-симуляция в «Вентилятор» ускоряется методом имплементации на ПЛИС аппаратного модуля, то в отличии от поведенческих RTL симуляторов для симуляции на «Вентилятор» имеет значение является HDL код синтезируемым или нет. Синтезируемым считается модуль написанный на Verilog или VHDL, код которого не содержит несинтезируемых конструкций (за исключением описанных в разделе *Верификационные конструкции поддерживаемые в синтезируемом коде*), то есть такой который целиком может быть имплементирован на ПЛИС. Ограничения на синтезируемость аналогичны любым коммерческим ПЛИС. Отчёт о том какие модули проекта были признаны несинтезируемыми и по какой причине можно найти в отчётах стадии Preprocessing (см. *Файлы логов и отчётов*).

4.2 Поддерживаемые HDL языки

«Вентилятор» поддерживает симуляцию проектов реализованных на различных языках предназначенных для аппаратного синтеза и верификации.

Синтезируемая часть проекта может быть реализована на:

- Verilog (стандарты IEEE 1364-2001/2005)
- VHDL (стандарты IEEE 1076-1987/1993/2002/2008)

Тестбенчи (несинтезируемая часть) могут быть реализованы на:

- Verilog (стандарты IEEE 1364-2001/2005)
- SystemVerilog (ограниченное подмножество стандарта IEEE 1800-2023[1])
- Python с использованием фреймворка cocotb[2]

[1] Подмножество поддерживаемое HDL симулятором Verilator, см. https://verilator.org/guide/latest/languages.html
[2] См. https://www.cocotb.org/

4.2.1 Верификационные конструкции поддерживаемые в синтезируемом коде

«Вентилятор» позволяет включать некоторые конструкции HDL языков предназначенные для верификации в синтезируемый код без потери возможности его аппаратного ускорения. В несинтезируемой части проекта возможно использование любых верификационных конструкций соответствующих стандарту языка.

Следующие конструкции будут функционировать на «Вентилятор» в синтезируемой части аналогично поведенческим симуляторам:

- Verilog-функции для управления сохранением вейвформы: $dumpfile и $dumpvars. См. *Вейвформы*.
- Функции для остановки симуляции по критерию ошибки: $assert в Verilog и assert в VHDL. При этом проверяемый критерий должен быть синтезируемым. Нарушение проверяемого критерия (аргумент assert становится ложным) приведёт к остановке симуляции с печатью соответствующего сообщения.
- Функции для безусловного завершения симуляции: $finish в Verilog и std.env.finish в VHDL.
- Функции для инициализации памяти: $readmemh/$readmemb в Verilog и file_io в VHDL. При этом допустимо лишь однократная инициализация массива из файла (аналогично синтезу на ПЛИС).
- Verilog конструкции primitive и table обычно используемые для задания стандартных ячеек будут интерпретированы как обычная логическая функция, что позволяет моделировать нетлисты стандартных ячеек без задержек.

Следующие конструкции не повлияют на синтезируемость модуля, но будут проигнорированы:

- Verilog-функции для остановки сохранения вейвформы $dumpon и $dumpoff.
- Функции для печати в терминал: $display в Verilog и report (без assert) в VHDL.
- Verilog конструкции specify для задания временных параметров и проверок в стандартных ячейках.

4.2.2 Перезапуск и параллельные симуляции

«Вентилятор» имеет функционал, связаный с экономией времени при работе с ним:

- Одновременно можно запускать параллельно несколько симуляций.
- Уже отработанный запуск можно перезапустить с определённой стадии. См. *Интерфейс симуляции*.

4.3 Ограничения

4.3.1 Отличия от поведенческих симуляторов

Важно понимать, что «Вентилятор» является не заменой, а дополнением к классическим поведенческим симуляторам призванным ускорить верификацию больших синхронных цифровых проектов. Основные отличия «Вентилятор» от поведенческих симуляторов:

- Отсутствует поддержки X-состояния сигнала (аналогично синтезу на ПЛИС).
- Симуляция с задержками в синтезируемой части (например нетлисты с SDF) невозможна.
- Возможно моделирование только синхронных дизайнов. Под синхронным дизайнов в данном случае подразумевается дизайн для которого максимальное количество уровней комбинационной логики между двумя триггерами или между вводом/выводом и триггером не превышает 10 тыс элементов. Подавляющее большинство цифровых RTL дизайнов отвечает этому требованию.
- В силу аппаратных особенностей реализации, «Вентилятор» имеет предел быстродействия моделирования составляющий около 100 кГц (тактов основной частоты моделирования в секунду).

4.3.2 Технические ограничения на размер дизайна

«Вентилятор» накладывает ограничение на размер моделируемого дизайна. Ограничения на один модуль составляют:

- Размер синтезируемой части симулируемого проекта не должен превышать 100 млн эквивалентных вентилей, где 1 эквивалентный вентиль соответствует 1 стандартной двухвходовой логической ячейке, либо 1 биту памяти.
- Симулируемый проект должен иметь не более 100 частотных доменов.
- Невозможно произвольное соотношение числа вентилей и логики в проекте.

Внимание: В рамках доступа к бета-версии ПАК накладываются дополнительные ограничения, см. *Ограничения текущей бета-версии*.

4.3.3 Mixed language симуляции

Симуляция на «Вентилятор» проектов содержащих HDL код одновременно и на Verilog и на VHDL возможна, однако есть ограничение:

- При включении модуля написанного на одном языке в модуль на другом языке допустимо использование только базовых логических типов (bit, integer, std_logic, std_logic_vector, string для VHDL). Передача сигналов других типов, таких как VHDL record через «границу» языка не поддерживается.

4.3.4 Ограничения текущей бета-версии

Текущая демонстрационная версия «Вентилятор», к которой предоставляется удалённый доступ имеет дополнительные ограничения:

- Размер дизайнов не должен превышать 10 млн эквивалентных вентилей на контейнер.
- Дизайн должен иметь только 1 частотный домен.
- Скорость сборки и симуляции ограничены из-за параллельной работы нескольких демонстрационных контейнеров.
- Возможно запускать до 4 параллельно запущенных симуляций на контейнер, если их суммарный объём не превышает 10 млн вентилей.

ГЛАВА 5

Интерфейс симуляции

5.1 Запуск симуляции

Для запуска симуляции на «Вентилятор» используется вызываемый из терминала хост-компьютера и находящийся в PATH скрипт *ventilator_run* . Скрипт принимает один аргумент - путь до JSON конфигурационного файла проекта, симуляцию которого требуется запустить. В случае если в папке проекта лишь один файл в формате JSON, достаточно указать путь до папки, этот файл будет использован в качестве конфигурационного.

При запуске симуляции в текущей рабочей папке будет создана (если ещё не существует) папка *runs*, в которой в свою очередь будет создана подпапка с именем вида <дата>_<время> (дата и время запуска скрипта *ventilator_run*). В эту подпапку будут сохраняться все временные файлы и логи запущенной симуляции. Также в папке *runs* создаётся симлинк *last* который всегда указывает на подпапку последней запущенной симуляции. Удаление не нужных данных из папки *runs* можно проводить как вручную, так и при помощи аргумента --*clean [количество]* к скрипту *ventilator_run*, где *[количество]* - необязательный аргумент определяющий сколько подпапок последних симуляций требуется оставить (по-умолчанию 0, то есть будут удалены все).

Прервать работу маршрута симуляции можно в любой момент при помощи сочетания клавиш Ctrl+C.

Существует возможность перезапустить ранее завершённую симуляцию. Для этого необходимо указать аргумент --*run_dir [папка маршрута]* к скрипту *ventilator_run*, где *[папка маршрута]* - путь до папки маршрута симуляции которую требуется перезапустить (по-умолчанию *runs/last*).

Подобным образом маршрут можно перезапустить маршрут с конкретной стадии. Для этого необходимо указать аргумент --*start_stage [номер стадии]* к скрипту *ventilator_run*, где *[номер стадии]* - номер стадии маршрута симуляции с которой требуется перезапустить (по-умолчанию 1). Если аргумент --*run_dir* не указан, то симуляция с аргументом --*start_stage* запустится в папке последнего маршрута (*runs/last*).

Пример запуска:

```
ventilator_run $ИМЯ_КОНФИГА.json --run_dir=runs/19700101_000000 --start_stage=3
```

5.2 Стадии работы маршрута симуляции

Маршрут симуляции на «Вентилятор» разделён на стадии соответствующие этапам подготовки проекта к запуску с использованием аппаратного ускорения (см. Рис. 1). В случае возникновения ошибки на одной из стадий, запуск будет остановлен и в терминал будет выведена информация об ошибке.

1. **Preprocessing**. На этой стадии производится первичная проверка конфигурационного файла и файлов исходного кода HDL, а также разделение модулей на синтезируемые и несинтезируемые. Остановка с ошибкой на этом этапе может произойти из-за ошибок в конфигурационном файле, либо синтаксических ошибок или неподдерживаемых конструкций в файлах HDL (см. *Поддерживаемые HDL языки*).

2. **Clustering**. На этой стадии производится разделение синтезируемой части дизайна на отдельные кластеры для имплементации в ПЛИС аппаратного модуля. Остановка с ошибкой на этом этапе может произойти из-за превышения максимального допустимого объёма дизайна. При успешном завершении стадии выводится оценка размера дизайна в эквивалентных вентилях.

3. **Implementation** / **Make binaries**. Параллельно исполняемые стадии в ходе которых производится имплементация синтезируемой части дизайна на ПЛИС аппаратного модуля и компиляция несинтезируемой части для исполнения на хост-компьютере. Остановка с ошибкой на данном этапе может произойти при использовании ошибочных или неподдерживаемых конструкций в несинтезируемой части дизайна, например в скриптах cocotb.

4. **Hardware init**. На этой стадии производится инициализация аппаратного модуля ускорения симуляции, в частности загрузка конфигураций ПЛИС для ускорения.

5. **Simulation**. На этой стадии происходит собственно симуляция проекта. В процессе симуляции терминальный вывод симулируемого проекта (результат выполнения $display в Verilog тестбенчах, информация о срабатывании $assert и др.) выводится в виртуальное «подокно» создаваемое в терминале (см. Рис. 3). Там же отображается информация о текущей скорости симуляции.

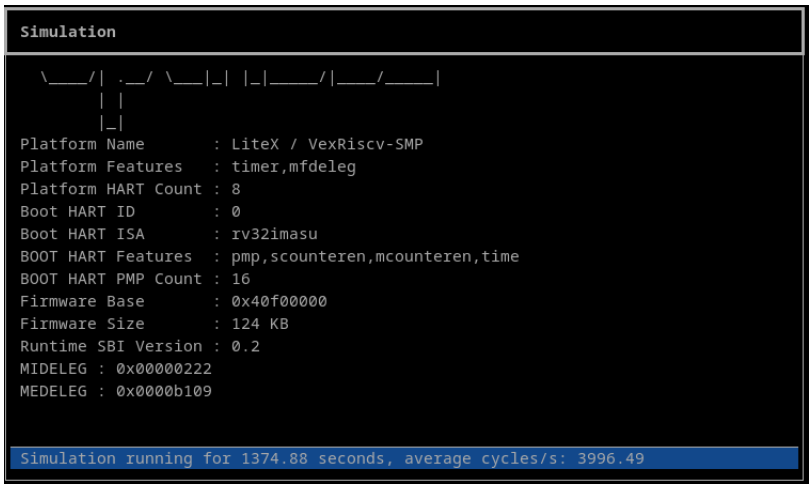

Рис. 3: Пример терминального вывода симулируемого проекта в консоль симулятора

5.3 Файлы логов и отчётов

Каждая из стадий работы маршрута симуляции создаёт логи, отчёты и временные файлы в своей подпапке папки симуляции внутри runs. Например все выходные файлы стадии Preprocessing будут располагаться по пути *runs/<дата>_<время>/1-preprocessing* .

Файлы логов и отчётов полезные для изучения пользователем:

1. `1-preprocessing/logs/paramsolver.log` - в случае возникновения ошибки предобработки одного из HDL файлов в этом файле можно найти более подробную информацию.
2. `1-preprocessing/logs/syntax_report.md` - в случае возникновения ошибки компиляции одного из HDL файлов в этом файле можно найти более подробную информацию.
3. `1-preprocessing/logs/synthesizability_report.md` - список модулей которые были признаны несинтезируемыми с причинами. Рекомендуется изучать этот отчёт при симуляции нового проекта чтобы убедиться что в этом списке присутствуют только ожидаемо несинтезируемые модули (например тестбенчи), так как это напрямую влияет на производительность симуляции.
4. `2-clustering/modules/sim_tops/<имя синтезируемого топ-модуля>/clustering_report.md` - отчёт о разбиении синтезируемой части проекта на фрагменты для отдельных ПЛИС с подробной информацией о размере проекта в эквивалентных вентилях. Этот отчёт может быть полезен для изучения в случае превышения проектом максимального допустимого размера для симуляции.
5. `3.2-make_bin/make.log` - в случае возникновения ошибки компиляции одного из файлов тестбенча в этом файле можно найти более подробную информацию.
6. `5-run_sim/testbench_output.log` - лог хранящий терминальный вывод тестбенча.

5.4 Вейвформы

Включение сохранения изменения части сигналов на вейвформу возможно как стандартными средствами Verilog (функции $dumpfile/$dumpvars), так и при помощи конфигурационного файла проекта (см. *Конфигурационный файл проекта*). Эти два способа можно совмещать друг с другом. Остановка и возобновление сохранения вейвформы при помощи Verilog функций $dumpoff и $dumpon не поддерживается, вейвформа сохраняется на протяжении всей симуляции.

Файл содержащий вейвформу будет расположен в папке *runs/<дата>_<время>/5-run_sim*. В случае активации сохранения вейвформы при помощи функции $dumpvars, имя файла будет определяться аргументом функции $dumpfile, а в случае использования конфигурационного файла файл вейвформы будет называться wave.fst или wave.vcd, в зависимости от выбранного формата. Формат vcd более распространён и поддерживается большим числом инструментов, однако крайне неэффективен в отношении размера получаемого файла. FST является открытым форматом для хранения вейвформ значительно превосходящим vcd по эффективности[1] и при этом поддерживается многими программными инструментами, например GtkWave.

Для автоматического открытия в графическом просмотрщике файла вейвформы после начала симуляции необходимо объявить конфигурационную переменную *SHOW_WAVE* в 1.

Рекомендуется ограничивать диапазон сохраняемых в файл вейвформы сигналов лишь необходимыми, так как избыточное число сохраняемых сигналов (например в случае сохранения всего проекта) может оказывать существенное негативное влияние на скорость симуляции.

Также при большом количестве сохраняемых сигналов генерация вейвформы может отставать от симуляции. По умолчанию генерация вейвформы останавливается сразу после завершения симуляции. Для генерации всей вейвформы необходимо объявить конфигурационную переменную *WAIT_WAVE* в 1.

[1] См. https://gtkwave.sourceforge.net/gtkwave.pdf Appendix F.

5.5 Переменные окружения

С помощью переменных окружения объявленных в момент запуска скрипта *ventilator_run* можно переопределять конфигурационные параметры задаваемые в конфигурационном файле проекта (см. *Конфигурационный файл проекта*). Например для включения графического показа вейвформы можно запускать симуляцию при помощи команды

```
SHOW_WAVE=1 ventilator_run $ИМЯ_КОНФИГА.json
```

ГЛАВА 6

Конфигурационный файл проекта

Для запуска RTL проекта на симуляцию в «Вентилятор» необходимо составить конфигурационный файл для этого проекта. Конфигурационный файл имеет формат JSON и содержит такие параметры как путь до исходных файлов проекта и др. Все пути в конфигурационном файле указываются относительно положения конфигурационного файла.

Абстрактный пример конфигурационного файла:

```
{
    "TOPMOD": "testbench",
    "SOURCES": [
        "tb/testbench.v",
        "tb/mem.v",
        "rtl/topmod.v",
        "rtl/wrappers.v",
        "rtl/generic_dpram.v",
        "rtl/vbuf.v",
        "rtl/getbits.v",
        "rtl/core/cpu_alu.vhd",
        "rtl/core/cpu_control.vhd"
    ],
    "ADD_FILES": [
        "memory.hex"
    ],
    "DEFINES": {
        "MODELINE_SIF": "",
        "TIMEOUT": "250ms",
        "SOURCE_STREAM_FILE": "\"tb/stream.dat\""
    },
    "TOPMOD_PARAMS": {
        "DATA_WIDTH": "8",
        "ID_WIDTH": "8",
        "ID_ENABLE": "1",
        "USER_WIDTH": "1"
    },
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    "VERILOG_INCLUDE_DIR": [
        "rtl/include",
        "tb"
    ],
    "DUMP_WAVE": {
        "testbench.topmod": 0
    },
    "SHOW_WAVE": 1,
    "WAVE_VCD": 0,
    "NO_WAVE": 0,
    "VERILOG_STANDARD": "1364-2005",
    "SYSTEM_VERILOG_STANDARD": "1800-2023",
    "STANDARDS": {
        "rtl/generic_dpram.v": "1800-2023",
        "rtl/getbits.v": "1800-2017"
    },
    "DEBUG_MODE": "verilator",
    "LOGLEVEL": 1,
    "LIB_INFO": {
        "core": [
            "rtl/core/cpu_alu.vhd",
            "rtl/core/cpu_control.vhd"
        ]
    },
    "ENVIRONMENT": {
        "PORTS": "4"
    }
}
```

Обязательные параметры:

`TOPMOD` - имя верхнего модуля (обычно тестбенч).

`SOURCES` - список входных файлов, содержит в себе исходные коды на HDL языках (Verilog или VHDL) и тестбенч на Verilog, Python (cocotb). Принадлежность файла к определённому языку определяется по расширению: .v - Verilog, .sv - SystemVerilog, .vhd/.vhdl - VHDL, .py - cocotb.

Необязательные параметры:

`ADD_FILES` - файлы не требующие отдельной компиляции, но требующие включения в проект (например файлы инициализации памятей и т.п.);

`DEFINES` - словарь определений препроцессора (defines) Verilog;

`TOPMOD_PARAMS` - словарь переопределения параметров верхнего модуля (parameter/localparam) Verilog;

`VERILOG_INCLUDE_DIR` - список путей для поиска заголовочных файлов Verilog;

`DUMP_WAVE` - словарь позволяющий указать сохранение вейвформы. Первый элемент пары - строка иерархического имени модуля начиная с которого необходимо включить дамп, а второй - число соответствующее иерархической «глубине» дампа (0 - бесконечность). Например пара *«top.my_module»: 0* включит дамп вейвформы в модуле top.my_module и всех вложенных в него модулях и их подмодулях. Про файл вейвформы см. *Вейвформы*;

`SHOW_WAVE` - при объявлении в 1, при запуске симуляции открывает просмотр генерируемой вейвформы в графическом просмотрщике GTKWave (по-умолчанию 0);

`WAVE_VCD` - включает сохранение файла вейвформы в стандартном формате vcd, по-умолчанию используется более компактный формат fst;

`NO_WAVE` - при объявлении в 1, отключает вейвформу (по-умолчанию 0);

`WAIT_WAVE` - при объявлении в 1, вейвформа генерируется полностью, иначе генерация останавливается при остановке симуляции (по-умолчанию 0);

`VERILOG_STANDARD` - стандарт для файлов с разрешением «.v» (по-умолчанию «1364-2005»);

`SYSTEM_VERILOG_STANDARD` - стандарт для файлов с разрешением «.sv» (по-умолчанию «1800-2023»);

`STANDARDS` - словарь с названиями файлов и стандартами для них (приоритетнее чем VERILOG_STANDARD и SYSTEM_VERILOG_STANDARD);

`DEBUG_MODE` - указывает уровень отладки, необходимо только для выявления внутренних ошибок проекта «Вентилятор», доступно: gdb, verilator, waveform, strace, None (по-умолчанию);

`LOGLEVEL` - указывает уровень логирования (по-умолчанию 0);

`LIB_INFO` - определяет VHDL-библиотеки, словарь с названием библиотеки и списком файлов входящих в неё;

`ENVIRONMENT` - словарь с переменными окружения, будут добавлены только те, которых нет в окружении.

ГЛАВА 7

Написание и подключение симуляторных моделей

«Вентилятор» поддерживает подключение симуляторных моделей написанных на Python (cocotb) и на C/C++ (интерфейсы System Verilog DPI и Verilator API).

7.1 C/C++ модели

Verilator позволяет проводить симуляцию Verilog/SystemVerilog, описывая тестбенч на C/C++. Verilog/SystemVerilog-модули преобразуются в Verilator-модели - C++ классы, которые затем подключаются в файл тестбенча. Verilator-модели имеют атрибуты, соответствующие портам Verilog/SystemVerilog-модуля, а фукнция `eval()` класса позволяет выполнить шаг симуляции модели.

Принцип работы «Вентилятора» заключается в вызове Verilator DPI функции (вызов C++ кода из Verilog кода) Verilator-моделей Verilog/SystemVerilog-модулей: код DPI функции `eval()`, отвечающий за выполнение шага симуляции, осуществляет обмен данными с ПЛИС. Таким образом, вычисления производятся на ПЛИС, а не на хостовой машине.

Чтобы адаптировать тестбенч, написанный на C++ в соответствии с API Verilator, не нужно вносить изменений в тестбенч; достаточно добавить файл (или несколько файлов) тестбенча в параметр `SOURCES` конфигурационного файла. При этом, особенности «Вентилятора» накладывают на тестбенч некоторые ограничения (см. раздел ниже).

В качестве примера рассмотрим C++ вариант теста *Последовательный умножитель (spm)*. Его конфигурационный файл выглядит следующим образом:

```
{
        "TOPMOD": "spm",
        "SOURCES": [
                "spm.v",
                "spm_tb.cpp"
        ]
}
```

Параметр `SOURCES` содержит файл с Verilog-модулем `spm.v` и C++ код тестбенча `spm_tb.cpp`. В данном случае параметр `TOPMOD` указывает не на модуль тестбенчба, а на `spm`, так как Verilog-модуля тестбенча не существует.

Отметим, что C++ код тестбенча может быть запущен и без «Вентилятора» исключительно средствами Verilator. В таком случае вычисления, необходимые для симуляции, будут производиться на хостовой машине.

7.2 Python модели

Фреймворк cocotb позволяет писать тестбенчи на Python и верифицировать VHDL/Verilog/SystemVerilog дизайны. В качестве симулятора cocotb позволяет выбрать Verilator; в этом случае cocotb с помощью Verilator генерирует C++ модель дизайна, управление которой происходит с помощью Python-кода cocotb. На этом и основывается принцип работы «Вентилятора» с Python-моделями: как и в случае C++ моделей, происходит вызов Verilator DPI функции с кодом для общения с ПЛИС. Таким образом, для запуска Python-тестбенча с помощью «Вентилятора», как и в случае C++ Verilator-тестбенча, дополнительная адаптация не нужна, достаточно добавить файл с Python-кодом в конфигурационный файл.

7.3 Ограничения на модели

Для того, чтобы сэкономить время создания Verilator-моделей и их компиляции, на вход Verilator подается не полный дизайн, а его урезанная версия. В частности, верхние синтезируемые модули заменяются на «пустышки», содержащие только объявления портов, необходимые для получения правильного интерфейса Verilator-модели. Таким образом, логика синтезируемых модулей не попадает в Verilator-модель; кроме того, Verilator-модель не содержит внутренних сигналов модулей, не являющихся портами верхних синтезируемых модулей. Это ограничение обходится автоматическим расширением портов и подключением к ним внутренних сигналов, однако это замедляет скорость симуляции и не работает в случае с API Verilator, поэтому не рекомендуется использовать доступ к внутренним сигналам синтезируемых модулей из тестбенча. Также не поддерживается вейвформа и доступ к внутренним модулям памяти.

Таким образом, чтобы тестбенч, написанный в соответствии с API Verilator, мог быть запущен с помощью «Вентилятора», в нем не должно быть обращений к внутренним сигналам синтезируемых модулей. Например, если `top` - верхний Verilog-модуль, а `p` - любой из его портов, то обращение `top.p` допустимо. Если же `s` - любой сигнал или порт модуля `mod`, а `top` содержит инстанс модуля `mod` с именем `M`, то обращение вида `top.M.s` недопустимо. Если `s` - сигнал модуля `top`, не являющийся портом, то обращение `top.s` также недопустимо.

ГЛАВА 8

Рекомендации по оптимизации симуляции на «Вентилятор»

Для достижения оптимальной эффективности использования «Вентилятор» необходимо учитывать некоторые рекомендации.

Скорость симуляции «Вентилятор» мало зависит от объёма проекта (в пределах допустимого объёма) и частоты переключения сигналов, однако может существенно зависеть от отношения количества синтезируемого кода к количеству несинтезируемого. При этом существует максимальный предел скорости симуляции, выше которого она не может подняться в силу аппаратных особенностей. Таким образом оптимальными с точки зрения производительности симуляции на «Вентилятор» проект должен:

- Иметь размер примерно от 1 млн эквивалентных вентилей. Более маленькие проекты симулировать нерационально, так как для них быстродействие «Вентилятор» скорее всего не превысит производительность поведенческого симулятора.
- Иметь существенную частоту переключений (toggle rate). Высокий toggle rate резко негативно влияет на производительность поведенческих симуляторов, но не влияет на производительность «Вентилятор». Таким образом проекты с высоким toggle rate (процессоры, DSP, криптографические блоки и др.) существенно выигрывают от симуляции на «Вентилятор».
- Состоять преимущественно из синтезируемого HDL кода. Несинтезируемый код, такой как тестбенчи, cocotb-тесты и DPI модели в случае «Вентилятор» исполняются на хост-компьютере и соответственно не будут иметь существенного выигрыша по производительности.
- Не требовать сохранения чрезмерного числа сигналов в файл вейвформы.

ПРИЛОЖЕНИЕ A

Приложение: Примеры дизайнов для симуляции

В настоящем приложении приводится краткое описание некоторых из поставляемых вместе с «Вентилятор» примеров проектов основанных на open source HDL коде. Более подробное описание дизайнов и рекомендации по запуску их симуляции на «Вентилятор» приводятся в Readme файлах в папках дизайнов.

Найти описанные примеры можно в папке *$HOME/ventilator/examples/* на хост-компьютере.

A.1 Последовательный умножитель (spm)

Минимальный дизайн призванный продемонстрировать пример конфигурационного файла для простейших случаев.

Источник: https://github.com/The-OpenROAD-Project/OpenLane/tree/master/designs/spm

A.2 Процессорное ядро NEORV32 (neorv32)

Небольшой дизайн представляющий из себя 32-битное RISC-V процессорное ядро микроконтроллерного класса реализованное на VHDL и сконвертированное в Verilog.

Источник: https://github.com/stnolting/neorv32

A.3 Процессорное ядро VexRiscv с загрузкой Linux (litex_vexriscvX_linux)

Дизайн представляющий из себя СнК с 32-битным 2-х (litex_vexriscv2_linux) или 8-ми (litex_vexriscv8_linux) ядерным RISC-V процессорным ядром VexRiscv. Пример демонстрирует возможность исполнения сложных тестовых сценариев с загрузкой ОС Linux на моделируемой СнК.

Источник: https://github.com/spinalhdl/vexriscv

A.4 Ядро open-source GPU OpenGlory (gpu_pipe_wb_cocotb)

Дизайн представляет из себя простое GPU ядро реализованное на VHDL с конмплексным тестбенчем с использованием фреймворка cocotb. Пример содержит два JSON конфигурационных файла, один демонстрирующий пример использования VHDL кода, другой - сконвертированного Verilog. cocotb тестбенч производит рендеринг тестовой модели и сравнение результата с эталоном. Предусмотрен вариант запуска с выводом растеризуемой модели в графическое окно, который может быть запущен при помощи скрипта run.sh в папке примера (требует графического подключения к хост-компьютеру Вентилятор, например при помощи VNC). Данный тест демонстрирует возможность реализации комплексных тестбенчей для ПАК Вентилятор.

Источник: https://github.com/egorxe/openglory

A.5 Криптовычислитель SHA-256 (nsha256)

Дизайн среднего размера представляющий из себя большое число криптографических ядер реализующих хеш-функцию SHA-256 и работающих параллельно по принципу криптографического майнера. Проект демонстрирует возможность моделирования с высоким toggle rate.

Источник: https://github.com/secworks/sha256