FPGA SYSTEMS

FPGA-Systems
magazine

FSM :: № GAMMA

Первый журнал о программируемой логике

V1.1

Страничка редакции

ПЛИС-культ привет, FPGA комьюнити!

Приветствую тебя на страницах народного творчества FPGA / RTL / Verification разработчиков.

К сожалению, выход третьего номера несколько затянулся, но всё же вышел. С чем всех нас я и поздравляю!

Такая инициатива, как журнал, полностью посвященного напралению разработки на ПЛИС оказалась, к моему большому сожалению, неподъемной инициативой, отнимающей не малое количество сил и времени.

В связи с этим, я бы хотел попросить откликнуться тех, кому инициатива действительно нравится и тех кто бы хотел увидеть новые выпуски: мне действительно нужна помощь. Задач по журналу много и продолжать его вести самостоятельно уже не представляется возможным. Если у тебя есть непреодолимое желание мне помочь, буду рад видеть тебя в команде.

На этом разрешите закончить монолог и пожелать приятного время препровождения на страницах этого выпуска.

До встречи на страницах четвертого номера журнала FPGA-Systems Magazine (FSM), информацию о выходе которого вы всегда сможете найти на веб странице нашего издания.

PS: если вы хотите написать о чем-то, но не можете определиться с темой, посмотрите этот список, где приведены 100+ тем, которые могли бы быть интересны читателям.

===

С уважением, вождь FPGA комьюнити

Коробков Михаил

Контакты

По всем вопросам обращайтесь в телеграм @KeisN13 или по электронной почте admin@fpga-systems.ru

Фролова Светлана Евгеньевна

Этап прототипирования в маршруте разработки СнК.

Цель этапа, составные части этапа и их реализация

click

обзор

Буренков Сергей Алексеевич

Стереокамера машинного зрения с поддержкой ИИ на базе FPGA и Arduino Portenta H7

click

реализация

Евгений Куклов, Сергей Балакший

Исследование возможностей чипа AG32

click

обзор

Александр Хлуденьков и команда «Криптозавры»

Криптопроцессор на FPGA

click

реализация

Ajeetha Kumari Venkatesan, Deepa Palaniappan, Hemamalini Sundaram, V P Sampath

FPGA Design and Verification using opensource Workflows

click

обзор

Лотник Виталий

Особенности разработки аппаратного LDPC кодера

click

реализация

Аверченко А.П.

Блочно схемные элементы в DEEDS

click

начинающим

Свинцов А.А.

Исследование эффективности верификации с использованием PyUVM и SystemVerilog-UVM

click

исследования

Шанаева М., Попов М.

Коты приходят в Versal

click

реализация

Бабаев Рашид Эльдарович

БИХ-фильтры: основные понятия, формы и расчет

click

исследования

Харабадзе Д.Э.

Реверсивный счётчик с семисегментным индикатором на ATF22V10

click

реализация

Артем Кашканов

Использование SystemRDL для проектирования регистровых блоков

click

туториал

Гуров В.В.

Процессор для Tang Nano 9K

click

туториал

Кудинов Максим

Вывод DVI с нуля под Yosys

click

туториал

Аноним

Удаленное программирование ПЛИС, с использованием программного пакета Xilinx ISE 14.7

click

tips & tricks

Туровский Дмитрий Николаевич

Заметки ПЛИСовода. Часть вторая.

click

tips & tricks

Этап прототипирования в маршруте разработки СнК. Цель этапа, составные части этапа и их реализация.

Фролова Светлана Евгеньевна

Начальник отдела прототипирования
АО НПЦ ЭЛВИС

Обсуждение и комментарии: link

Наша компания занимается разработкой большого количества СнК. В основном, это процессорные микросхемы с большим количеством периферии. Ярким примером является процессор СКИФ, который сейчас широко применяется в отечественных устройствах. В этой статье пойдет речь о построении этапа прототипирования в маршруте разработки больших процессорных СнК из опыта АО НПЦ «ЭЛВИС».

Также, в этой статье, исходя из аудитории сообщества FPGA-systems, хотелось бы обратить внимание на отличия в разработке проектов для FPGA и ASIC (в русском переводе: ПЛИС и СнК – система-на -кристалле).

Маршрут разработки СнК можно грубо представить следующим образом:

ТЗ – разработка архитектуры – разработка RTL – функциональная верификация – прототипирование – разработка топологии – изготовление – тестирование полученных образцов.

Для процессорных СнК в маршруте присутствует также разработка программного обеспечения, причем желательно, чтобы большая часть программ: драйвера устройств, компиляторы, ОС, были подготовлены до получения изготовленной микросхемы.

Стадия прототипирования начинается тогда, когда RTL достиг достаточного уровня зрелости для запуска определенных сценариев. До этого RTL должен пройти основные тесты на этапе функциональной верификации, чтобы на прототипе, где поиск багов сложнее, чем на симуляции, меньше тратить времени на поиск простых ошибок.

Основной задачей прототипирования по классическому подходу является создание и предварительный запуск программного обеспечения будущей микросхемы. Если ПО будет создано и протестировано заранее, это существенно уменьшает длительность этапа освоения («BringUp») микросхемы и облегчает работу с «живой» микросхемой.

Помимо основных целей, в начале работы на прототипе обычно выявляются некоторые системные ошибки, которые не были предусмотрены в проверке на симуляции RTL (по взаимодействию ресетов,

спящий режим, ошибки в правах доступа с различных интерфейсов к внутренним блокам и т.д.)

Также, по опыту работы нашей компании, иногда к прототипам возвращаются для проверки определенных режимов работы уже на стадии bring-up, когда при запуске уже готового чипа возникает какая-то ошибочная ситуация.

Составные части этапа прототипирования

Этап прототипирования – это не просто разработка проекта на FPGA. Этап включает в себя несколько аспектов.

Первый из них – это подбор аппаратной платформы. На базе анализа сценариев использования будущей СнК и изучения состава интерфейсов проекта принимается решение об использовании существующей платформы прототипирования или создании или покупке новой платформы под проект

Далее, под существующую платформу исходный RTL модифицируется. Вместе с этим, продумываются способы отладки проекта на прототипе и добавляются дополнительные RTL-блоки для управления, отладки и сбора различных трасс. Эти блоки разрабатываются специально для прототипа.

Для того, чтобы созданные с использованием прототипа тесты и ПО могли быть использованы для тестирования и работы с готовой микросхемой, должны быть сделаны программные «прослойки», которые позволят переносить тесты с прототипа на симуляцию в функциональную верификацию в случае нахождения сбойной ситуации.

Следующий аспект – это организация доступа пользователей прототипа для работы с платформой прототипирования. Обычно мы организовываем удаленный доступ к хост-компьютеру прототипа с системой бронирования и поддержкой инженерами-разработчиками прототипов.

И последнее - это поддержка работы пользователей прототипа и анализ проблемных ситуаций при запуске тестов на прототипе.

Теперь рассмотрим подробнее каждый аспект:

1. Подбор платформы прототипирования

Идеальным вариантом является платформа, в которой есть много логики и много интерфейсов, в том числе высокоскоростных.

Проекты на миллионы и миллиарды вентилей требуют сложных и дорогих платформ прототипирования, самодельных или покупных. Такие платформы позволяют заложить в прототип всю цифровую логику СнК. Мы стараемся проектировать и использовать одну мощную платформу с большим объемом логики для нескольких проектов, закладывая в нее интерфейсы, используемые в большинстве проектов, такие, как: в первую очередь, отладочные интерфейсы процессоров, DDR-память, QSPI, I2C, PCIe, Ethernet. Однако минусом таких платформ является низкая скорость FPGA-проекта: до 10 Мгц. Провести тестирование со вводом данных с одного высокоскоростного интерфейса, обработку на большом количестве цифровой логики и вывод потока на другой интерфейс не всегда может получиться. Поэтому сценарии, использующие высокоскоростные интерфейсы, приходится использовать с некоторым изменением.

Для запуска разных сценариев и задействования различных интерфейсов используются разные платформы прототипирования. Один и тот же проект мы

прототипируем, используя различные платформы, разделяя проект на подмножества по сценариям проверки.

2. Создание проекта прототипа.

Поскольку эта статья предназначена для сообщества плисоводов, хочется остановиться на этом аспекте поподробнее.

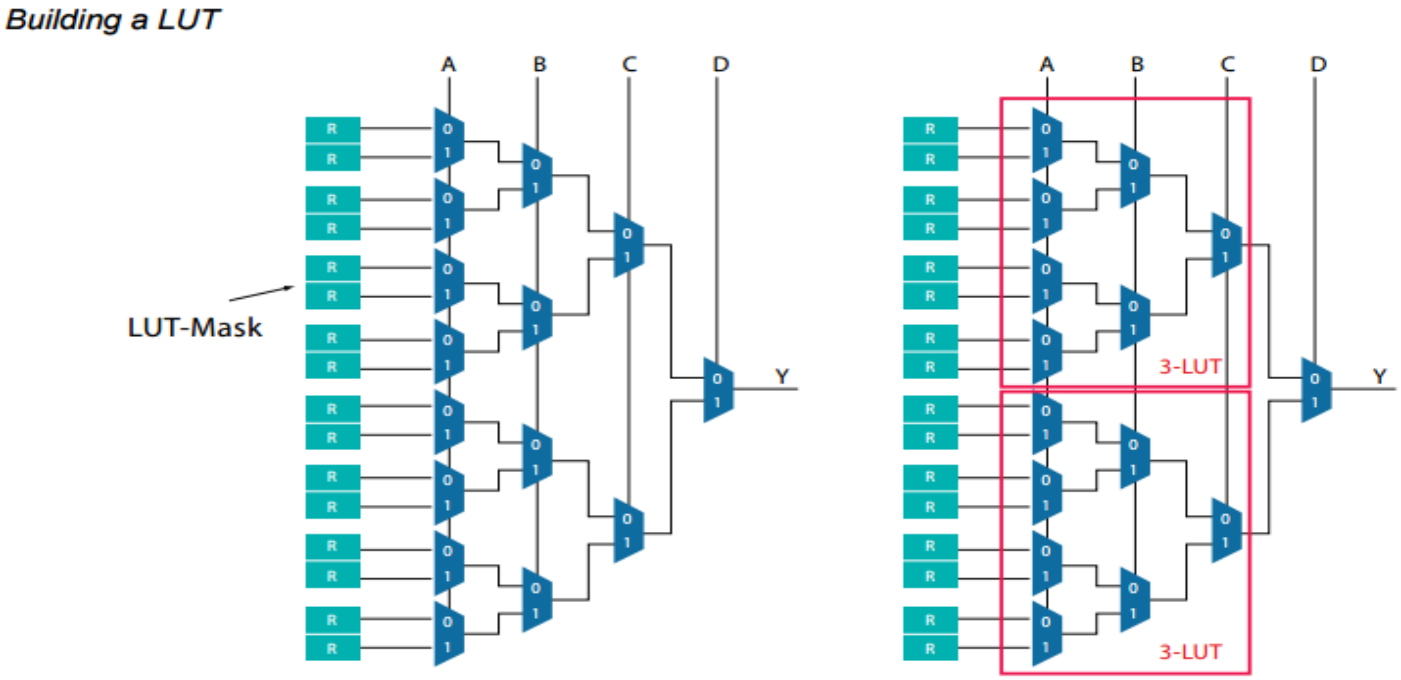
Существует значительная разница в разработке проекта для ПЛИС и для СнК. Язык Верилог один, а подход к схемотехнике существенно отличается.

ПЛИС разрабатывать существенно легче, поскольку вендор и САПР ПЛИС уже позаботились о многих вещах в имплементации верилог-кода. Многие начинающие разработчики могут о них даже не знать.

ПЛИС представляет собой матрицу регулярной структуры, состоящую из логических блоков: CLB, блоков памяти, hard IP высокоскоростных интерфейсов DDR, PCIe. Между ними проходят сигнальные линии и отдельные выделенные линии для частотных сигналов. Логическая функция формируется программированием линий межсоединений и логических блоков при прошивке ПЛИС bit-файлом.

Программирование логических функций: В сообществе fpga-systems много обсуждений на тему того, как реализовать ту или иную математическую функцию. Опытный разработчик ПЛИС знает, как описать функцию, чтобы она максимально плотно и оптимально «легла» на структуру LUT.

Рассмотрим структуру LUT одного из типов ПЛИС. LUT фактически представляет собой SRAM, содержимое которой хранится в конфигурационной памяти. В LUT можно имплементировать любую функцию по таблице истинности. На рисунке ниже показан пример имплементации 3-входового LUT по заданной функции.



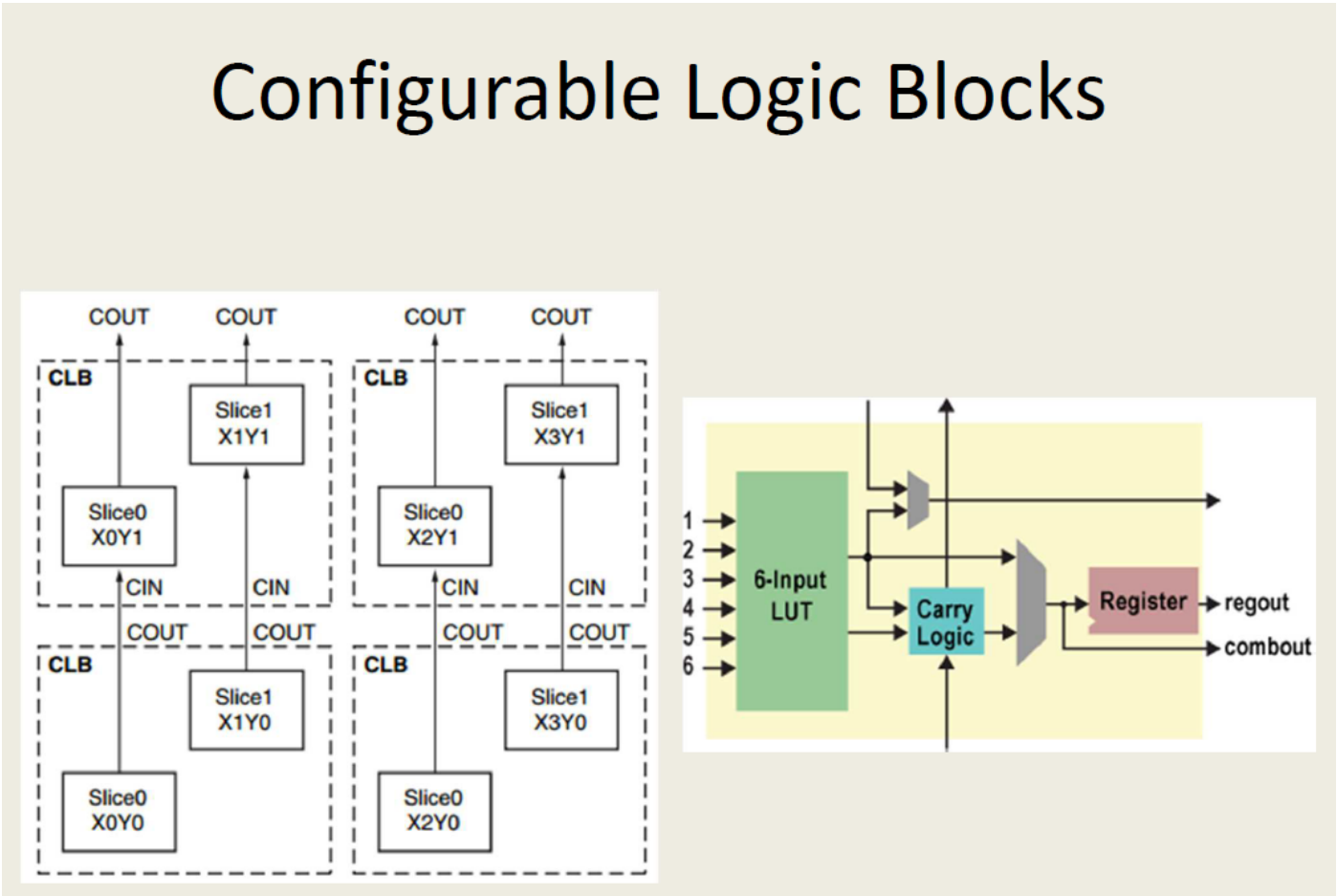
LUTы объединены в слайсы, состоящие из LUT, логики переноса и D-триггера. Слайсы объединены в CLB.

Трассы сигналов между CLB оптимизированы по задержкам. RTL проекта раскладывается по этим ячейкам в зависимости от необходимого числа триггеров и логики. Если число входных сигналов превышает возможности LUT, логика делится на два LUT, и так далее. В результате мы получаем проект с не полностью заполненными ячейками.

Здесь голубым цветом показаны занятые ячейки. Видно, что в слайсе заняты в основном LUTы, а триггеры мало задействованы.

В ASIC топология не является «квадратно-гнездовой». Она набирается из стандартных ячеек. С одной стороны, получается более плотная логика на единицу площади, с другой стороны, цепи сигналов неравной длины и, соответственно, с разными задержками.

Необходимо продумывать объем комбинационной логики между триггерами, взаимное расположение и количество блоков, связанных общей логикой. Достаточно частый случай, когда приходится переписывать RTL А по результатам топологии.



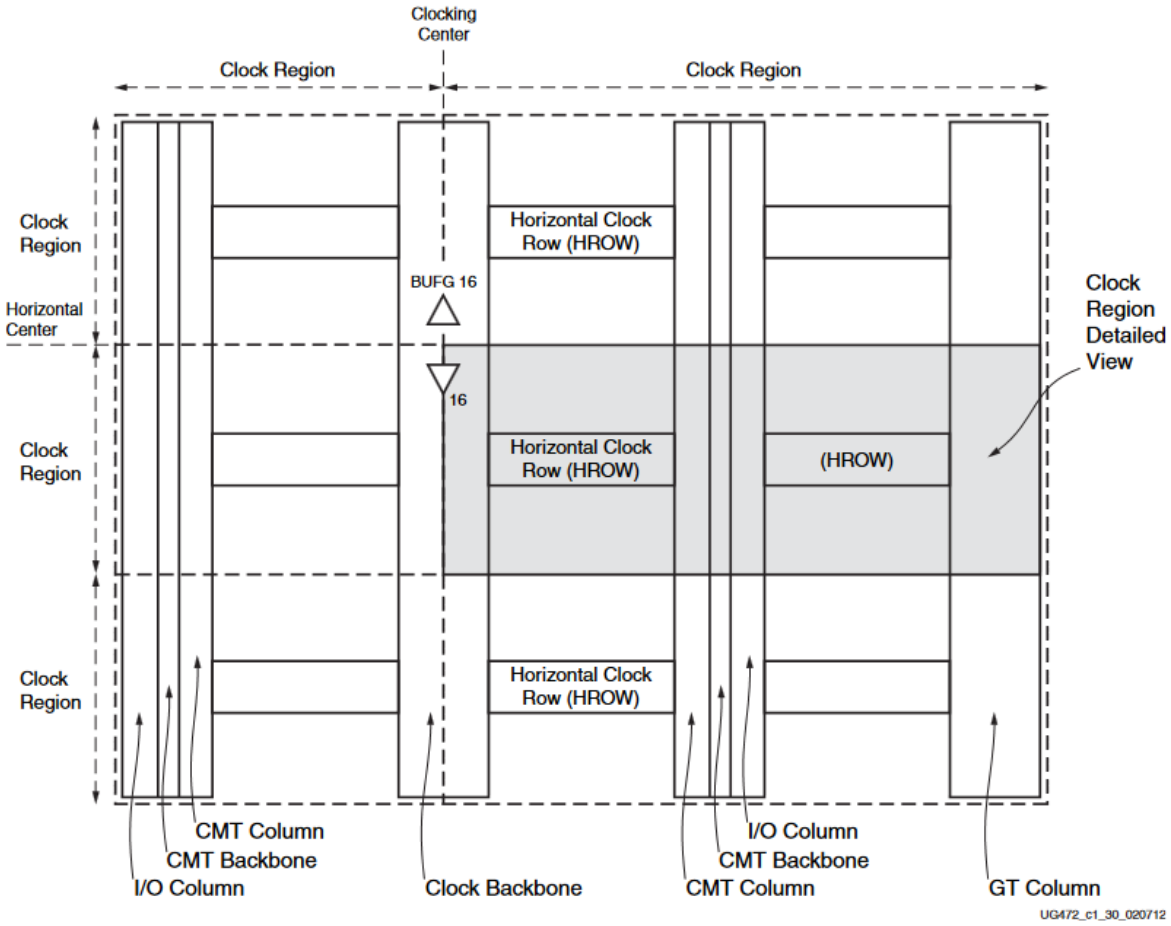
Разводка частот в ПЛИС и требования к частотам в СнК.

На ПЛИС вендоры самой внутренней структурой ПЛИС уже решили много задач по распространению частот. Пример из FGPA Xilinx 7 серии: «каждое устройство 7 серии имеет 32 глобальные частотные линии, которые могут тактировать и обеспечивать управляющими сигналами все последовательные ресурсы во всем устройстве. Каждый частотный регион может поддерживать до 12 этих глобальных частотных линий, использующих 12 горизонтальных частотных линий в этом частотном регионе»

Поскольку вся структура ПЛИС регулярная, с регулярной системой разводки частот по выделенным линиям, то нет проблем с разводкой частот даже на достаточно больших блоках. В отличие от ПЛИС, в СНК необходимо при разработке RTL продумывать частотные регионы: какие частоты блока будут передаваться в другой блок, как разделить дизайн на частотные домены для разработки топологии кристалла, количество потребителей каждой частоты.

Физические уровни интерфейсов(PHY)

Для высокоскоростных интерфейсов, таких, как PCIe, Ethernet, DDR, помимо логической стыковки по сигнальным линиям есть блоки физического сопряжения – PHY. Эти блоки технологически зависимы. Блок в ПЛИС является hard IP и не может быть заменен на другой. В некоторых интерфейсах и ПЛИС блоки представляют собой hard IP вместе с контроллером интерфейса, в некоторых – отдельные, только физические IP. Многие PHY FPGA имеют стандартные интерфейсы с контроллером, однако не все. Например, физический уровень DDR имеет нестандартный интерфейс и не может быть



7 Series FPGA High-Level Clock Architecture View

состыкован с другим контроллером, кроме как IP-контроллер от FPGA. При разработке на ПЛИС обычно используются блок-дизайны или hard IP вставляется просто «куском». Для ASIC требуется более глубокое понимание интерфейсов для грамотной стыковки физического уровня.

Контроллеры интерфейсов и шин

Многие ПЛИС имеют достаточно широкий спектр встроенных IP-блоков «на все случаи жизни». Блок-дизайн из готовых кирпичиков всегда под рукой. Легко можно построить любую систему. В ASIC, конечно, также используются некоторые покупные IP-блоки, но их интеграция в систему – в руках и на совести разработчика RTL. Разные IP имеют различные интерфейсы, различные требования к частотам и состыковать их – серьезная задача.

Процессорные ядра.

Тут все понятно: встроенные процессоры ПЛИС имеют всю инфраструктуру для встраивания в проект и всю программную среду разработки с драйверами интерфейсов. В случае ASIC все надо продумывать самим.

Отладка

ПЛИС имеет огромные возможности отладки через отладочные интерфейсы и средства отладки, к примеру, Xilinx hardware manager – chipscope. Это сильно расхолаживает разработчиков: что-то придумали, быстро прошили, посмотрели. Не понравилось, перешить не проблема. В ASIC цена запуска микросхемы на фабрику огромная, поэтому цена ошибки тоже огромная. Это и есть причина, почему проекты ASIC имеют множество стадий верификации, в том числе этап прототипирования. Для того, чтобы изготовленный ASIC можно было «пощупать» через какую-то систему, была придумана система DFT – «design-for-test» и BIST – build-in-self-test. Эти системы обязательно встраиваются в любой большой ASIC, и это отдельная сложная работа.

В нашей компании был опыт переноса проектов отдельных блоков с FPGA на ASIC, но опыт показал, что этот путь очень затратный и невыгодный. Прежде всего, из-за того, что очень много времени ушло на переделку исходного кода для того, чтобы его можно было интегрировать в ASIC.

Типичные ошибки разработчиков ПЛИС:

- Отсутствие ресета и начальной инициализации триггеров. В ПЛИС существует отдельный блок, вставляемый автоматически, который обеспечивает обнуление всех триггеров по включению. Если ресет как вход блока отсутствует и внутренние триггера в начальном состоянии не определены, это приводит к непредсказуемому поведению блока.
- Ошибки пересинхронизации частотных доменов. САПР FPGA, конечно, выдает предупреждения, но кто их читает… В ASIC такое не пройдет. Системы контроля правил проектирования, а это САПРы, строго проверяют все правила, не пропустят такой проект.
- Латчи в дизайне. Использование латчей в дизайне - это исключительный случай. Часто синтез в латчи происходит из-за ошибки в RTL, если триггер неправильно описан в верилоге.
- Constraints – файлы ограничений. Очень часто разработчики ПЛИС, особенно те, кто занимается преимущественно проектированием линейных арифметических операций, плохо представляют себе констрейны для синтеза. В лучшем случае могут написать

констрейн, определяющий размер частоты. Для синтеза в ASIC и последующих работ по топологии констрейны имеют очень большое значение. Например, если не указать multicycle, то топологи будут пытаться сделать топологию в один такт на пути, где арифметическая операция выполняется несколько тактов, и в итоге получат максимальную возможную частоту в несколько раз меньше возможной.

Из-за всех вышеперечисленных причин приходит вывод, что проект для ASIC изначально пишется специализированно под ASIC. Переделывать проект из FPGA на ASIC крайне непродуктивно.

Задача создания прототипа: с минимальным количеством изменений перенести исходный код ASIC на ПЛИС и проверять именно этот код. Отсюда множество ограничений для разработчиков прототипов:

- Разработчики RTL ASIC и разработчики прототипов должны быть разными командами. Это принципиально. Общеизвестный факт: автор может выявить 85% своих ошибок. Остальные 15% должны выявить команды верификации и прототипирования.
- Нельзя модифицировать логику работы блоков: вставлять дополнительные регистры, улучшать тайминг через модификацию исходного кода. Если видно, что некоторые логические пути очень длинные, лучше дать об этом знать разработчикам RTL, это им пригодится при передаче проекта в топологию.
- Логика сложных IO-ячеек также должна быть перенесена по возможности без изменений.
- Все изменения кода при «урезании» проекта или добавления дополнительных блоков не должны влиять на логику работы и адресное пространство проекта
- Исходный код модифицируется под готовую платформу прототипирования. Поэтому расположение выводов фиксированное, входные частоты – также фиксированные от генератора на плате.

Исходя из вышеперечисленного, часть привычного инструментария САПР FPGA не может быть использована при разработке прототипа. Для отладки тайминга остается фактически только инструмент partition и игра некоторыми констрейнтами, изменение стратегии синтеза и имплементации.

Необходимая модификация RTL ASIC

Часто прототипирование выглядит так, как на картинке: попытка «утрамбовать» в FPGA большой проект.

Для конкретной платформы прототипирования приходится вырезать из кода некоторые блоки и-за нехватки места в FPGA. Далее идет модификация кода:

- замена PLL ASIC на PLL FPGA,
- модификация частот. Сталкивались с нехваткой частотных линий в ПЛИС, если в проекте много разных частот. Приходится упрощать частотные схемы проекта.

- замена технологических памятей на памяти FPGA или синтезируемые поведенческие модели памятей
- вырезание PHY высокоскоростных интерфейсов и замены на PHY FPGA.
- Иногда приходится заменять контроллеры ASIC на контроллеры ПЛИС

Следующая существенная часть разработки прототипов – это разработка и вставка дополнительных блоков для прототипирования. Необходимость и состав блоков определяется следующими причинами:

- Необходимость программного доступа к регистрам устройства;
- Необходимость введения дополнительных регистров для программного управления статическими внешними входами ASIC и программного контроля состояния событийных внешних выходов;
- Нехватка интерфейсов на платформе прототипирования: создаются блоки генераторов для имитации данных с отсутствующего интерфейса или блок приемника данных;
- При работе с определенными сценариями нужно мониторить и сохранять определенные трассы сигналов внутри проекта.

Теперь мы сделали прототип. Перед передачей его в работу команде программистов необходимо проверить работу его блоков, как вставленных нами, так и блоков проекта. Это требует серьезной работы по изучению прототипируемого проекта, некоторых навыков по написанию тестовых программ/скриптов и умению работать с отладчиками, как отладчиками процессоров, так и аппаратным отладчиком ПЛИС. Если исходный РТЛ «сырой», не прошел еще достаточного уровня функциональной верификации, то на этой стадии при тестировании проекта мы находим ошибки исходного RTL. Для локализации ошибок требуется «разработческая» квалификация со знанием работы внутренних интерфейсов СнК, пониманием процессов, происходящих в проекте.

3. «Прослойки»

Для работы с прототипом наших больших СнК необходимо позаботиться о воспроизводимости проблемных ситуаций. Для того, чтобы созданные с использованием прототипа тесты и ПО могли быть использованы для тестирования и работы с готовой микросхемой, должны быть сделаны программные «прослойки», которые позволят переносить тесты с прототипа на симуляцию в функциональную верификацию в случае нахождения сбойной ситуации.

4. Организация доступа к прототипу

Дальше, после предварительной работы с прототипом нашими средствами, мы передаем прототип для работы программистам. Для этого создается инфраструктура для удаленного доступа к прототипам, на хост-компьютер прототипа устанавливается необходимое ПО, делаются таблицы бронирования.

Разработанный прототип используется с несколькими целями:

- разработка драйверов устройств;
- разработка операционных систем;
- отладка загрузочных алгоритмов;
- прогон основных сценариев работы будущей микросхемы.

При работе пользователей самым сложным и трудоемким является процесс

поиска причин ошибок или зависания при запуске ими тестов. Достаточно сложно понять, где находится причина: в ошибке программирования или ошибке в аппаратной реализации. Здесь приходится вести достаточно сложную совместную работу, требующей понимания того, что происходит в программе и как это «ложится» на аппаратуру, в каких блоках может быть проблема.

На финальной части прототипирования к работе подключаются пользователи, которые отлаживают уже прикладные програмные задачи. Здесь может быть выявлено неудобство работы с нашей СнК в части программирования. Был случай, когда пользователи на этом этапе инициировали изменение внутри архитектуры будущей СнК. Это все позволило оперативно скорректировать проект до отправки на фабрику.

Когда СнК уже изготовлен и идет процесс его освоения («bring up»), прототип может пригодиться для проверки и анализа проблем, которые возникают при программировании чипа на отладочных платах.

Заключение – резюме

Прототипирование СнК – сложный этап в разработке больших СнК. Процесс прототипирования состоит из нескольких аспектов: подготовки платформы прототипирования, подготовки прототипа, тестирования прототипа, создания среды для работы программистов, анализа проблем при запуске тестов. Подготовить ПЛИС-проект для прототипа – задача, состоящая из многих компонентов. Разработка проекта на FPGA и разработка прототипа – не одинаковые задачи. Разработчик прототипов должен быть не просто FPGA-инженером, а инженером-разработчиком РТЛ и отчасти программистом, понимающим все блоки прототипируемого СнК, понимающим сценарии тестирования и вникающим в запуск тестов на прототипе.

Список литературы

1. https://electronics.stackexchange.com/questions/169532/what-is-an-lut-in-fpga
2. https://users.ece.utexas.edu/~mcdermot/arch/articles/Zynq/ug472_7Series_Clocking.pdf
3. С.Фролова, Ф.Путря. Создание многофункциональной и мультипроектной платформы прототипирования на базе комплекта HAPS компании Synopsys. Часть 1 https://www.electronics.ru/journal/article/7404
4. Фролова С.Е., Янакова Е.С. Методы достижения максимальной эффективности платформы прототипирования высокопроизводительных систем на кристалле на задачах искусственного интеллекта https://elibrary.ru/item.asp?id=43004623

Исследование возможностей чипа AG32

Евгений Куклов

Сергей Балакший

Обсуждение и комментарии: link

Аннотация

В данной работе будет представлено описание исследования и возможностей проектирования чипа AG32VF407. Этот чип входит в семейство AG32 и представляет собой систему на кристалле с 32-х битным процессором RISC-V, развитой периферией с высокоскоростными устройствами, подключенными к шине AHB, также с относительно медленными устройствами, подключенными к шине APB. Интересной особенностью данного чипа является встроенная программируемая логика FPGA объемом 2K LUT. Это обстоятельство несомненно представляет интерес для исследования возможностей программирования как самого чипа, так и возможностей по проектированию устройств на основе этого кристалла.

Обзор устройства на кристалле AG32

Что внутри?

Кратко это выглядит так:

- Ядро RISC-V с максимальной скоростью 248 МГц
- До 1 Мбайт Flash памяти
- SRAM 128 КБ
- 1 x CAN2.0, 2 x SPI, 2 x I2C, 5 x UART
- 2 x базовых таймера, 5 x таймеров расширенного управления
- 2 x сторожевых таймера
- 1 x системный таймер
- 10/100 Ethernet MAC
- Поддержка USB FS+OTG
- 3×12-битных, 1,0 MSPS АЦП: до 16 каналов и 3 MSPS в режиме тройного чередования
- 2×10-и битных ЦАП
- Два аналоговых компаратора rail-to-rail
- Универсальный DMA
- RTC
- 2K LUT FPGA, 4 блока M9K общим объёмом 16 Кбит.

- 128-и битный уникальный идентификатор
- Питание 3,0-3,6 В для приложений и входов/выходов. Режимы сна (Sleep), остановки (Stop) и ожидания (Standby). Питание VBAT для RTC
- Кристалл выполнен в корпусах QFN32, LQFP48, LQFP64, LQFP100. Имеет до 78 пользовательских портов ввода/вывода.

1.1.1 Управление часами, сбросом

- POR, PDR
- 4-26 МГц кварцевый генератор
- Внутренний 20 МГц генератор
- 32 кГц генератор для RTC
- Внутренний 40 кГц генератор
- Режим отладки — последовательный интерфейс отладки (SWD) и интерфейсы
- JTAG

В завершение обзора приведу общую структурную схему кристалла, отражающую его архитектуру.

Весь набор периферии одинаков для кристаллов всего семейства.

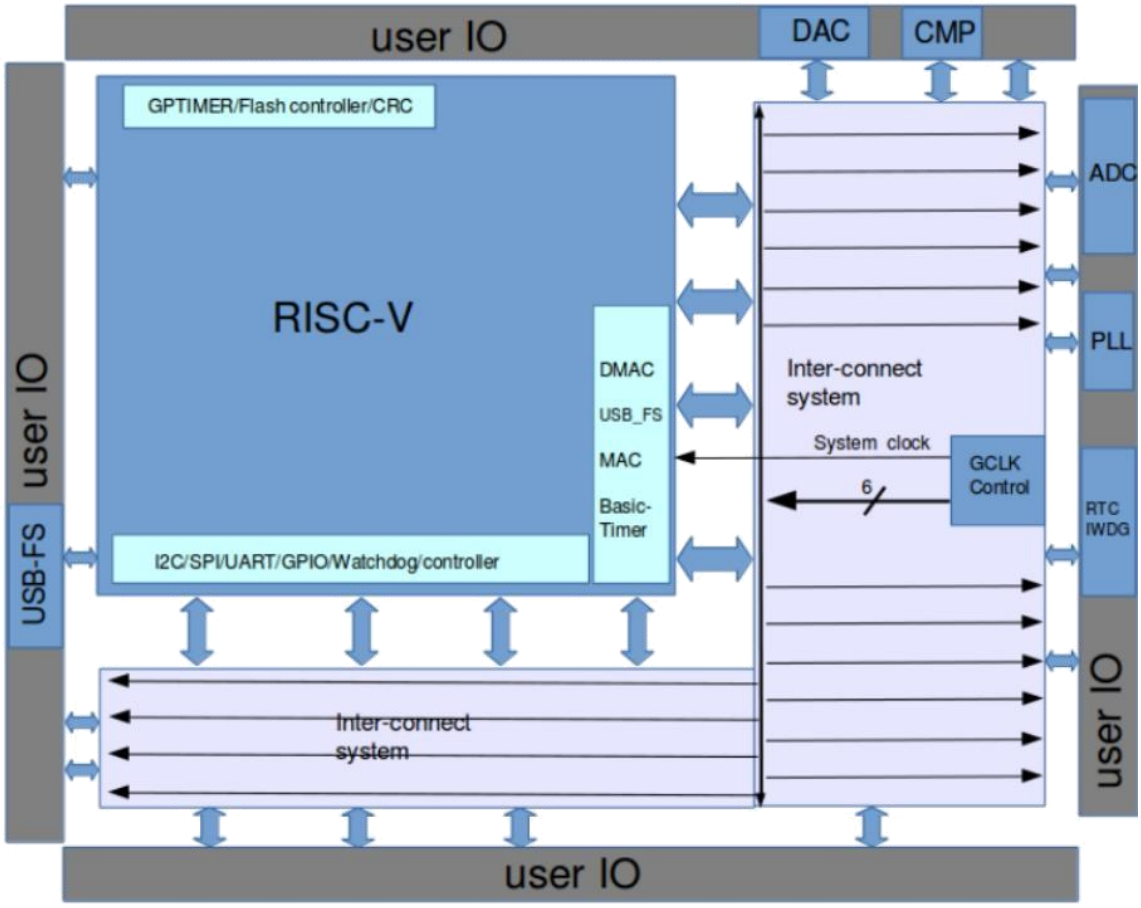


Рис. 1: Архитектура кристалла.

Инструменты разработки

1. Для работы с MCU производитель рекомендует использовать VScode и PlatformIO для создания среды разработки, при этом производителем поставляется SDK AgRv_pio для этой среды, с предустановленным Python версии 3.10.
2. Файлы дизайна, написанные на верилоге для FPGA, обрабатываются с использованием Quartus, на основе чипа Cyclone IV поскольку собственного чипа для AG32 в Quartus нет. Обработка пользовательской логики завершается только компиляцией с получением выходного файла .vo.
3. С помощью инструмента Supra дизайн, отражённый в файле .vo:

    a) преобразуется под конкретный чип AG32 в загрузочный файл .bin.

    b) может быть загружен в флэш память для фиксации логики FPGA.

Варианты использования

1. Поскольку, как уже отмечалось выше, в чипе AG32 наряду с микроконтроллером (MCU) имеются встроенные логические элементы FPGA, то при использовании чипа AG32 есть три варианта:
2. Использовать только часть, относящуюся к микроконтроллеру (MCU) независимо от FPGA, при этом FPGA находится в неактивном состоянии, включены лишь линии подачи тактовой частоты.
3. Использовать только часть FPGA, при этом MCU находится в неактивном состоянии.
4. Использовать совместно MCU и FPGA при взаимодействии через шину AHB обеспечивая самую высокую производительность.

Базовое понимание

Прежде чем переходить к практике, необходимо ознакомиться с некоторыми особенностями кристалла AG32.

1. Чип AG32 состоит из двух частей: MCU и FPGA. Эти две части независимы друг от друга (программируются отдельно и загружаются отдельно (создаётся два бинарных файла)), но их можно соединить друг с другом используя внутренние коммуникационные линии. Прежде чем чип начнёт работать, в него необходимо записать оба бинарных файла: как для MCU, так и для FPGA.
2. Подключение MCU и FPGA к внешним контактам (PIN) настраивается через файл .ve. В AG32 внешние контакты всех GPIO и большинства периферийных устройств не фиксированы. Соответствие необходимо указать в файле .ve. В .ve помимо настройки ассоциации между GPIO и PIN, вы также можете настроить ассоциацию сигналов между MCU и FPGA.
3. Часть проекта FPGA автоматически генерируется командой Vscode «Prepare LOGIC». Примечание. Работа FPGA зависит от проекта Vscode. Входной сигнал верхнего модуля в FPGA связан с проектом MCU через файл согласований .ve.
4. Кофигурации проекта:

    a. Если в проекте планируется использовать только MCU и не планируется использовать FPGA для создания пользовательской логики, конфигурацию проекта можно выразить как «использовать логику по умолчанию».
    Примечание. При этом в файле .ve необходимо задать конфигурацию тактовой частоты и соединение сигналов MCU с пинами ввода/вывода.

b. Если в проекте используются и MCU, и FPGA, или только FPGA конфигурация называется — «использовать пользовательскую логику». При этом в файле .ve кроме конфигурирования тактовой частоты, необходимо настроить корреляцию сигналов для трёх ситуаций:

   i. между MCU и PIN;

   ii. между MCU и FPGA;

   iii. между FPGA и PIN.

   Разработчик добавляет пользовательскую логику и компилирует проект в целом для MCU и FPGA.

   В любом случае дизайн проекта MCU опирается на дизайн FPGA будь то “логика по умолчанию”, либо случай “использования пользовательской логики”. После компиляции и создания соответствующих файлов .bin кристалл заработает, когда оба файла будут загружены в него.

c. При использовании в проекте только FPGA вы получаете только один файл .bin, кратко процесс проектирования можно описать так:

   i. Определите конфигурацию пинов в файле .ve;

   ii. Используйте команду подготовки “Prepare LOGIC” в vscode для создания структуры проекта FPGA;

   iii. Используйте “Quatus”, чтобы открыть созданный проект, добавить свой собственный логический код и, наконец, преобразовать проект;

   iv. Используйте “Supra”, чтобы открыть преобразованный проект и скомпилировать проект в конечный .bin файл. Загрузите его в чип.

Вышеприведенное описание представляет собой упрощенное описание, предназначенное прежде всего для того, чтобы дать общее впечатление.

Связь между MCU, FPGA и внешними контактами

Взаимосвязь между MCU и внешними контактами Формат определения

```
function_name_mcu PIN_XX # комментарий
```

где

- function_name_mcu - “имя функции микроконтроллера” находится в начале строки;
- PIN_XX — следует за function_name_mcu;
- # комментарий.

Например, можно определить GPIO для внешнего контакта - “GPIO4_3 PIN_32”, или определить последовательный порт 0 для внешнего контакта - “UART0_UARTRXD PIN_69”.

```
GPIO4_3      PIN_32 # LED3

UART0_UARTRXD PIN_69
```

где символ # отделяет комментарий от текста программы.

Связь между FPGA и внешним контактом

Формат определения:

```
signal_name_fpga PIN_XX:Direction # комментарий
```

где,

- signal_name_fpga имя сигнала fpga, определённое пользователем;
- PIN_XX - “идентификатор внешнего

контакта";

- Direction - направление, одно из "OUTPUT", "INPUT" или "INOUT".

Например, определение вывода для внешнего контакта

```
LED_D2 PIN_31:OUTPUT

LED_D3 PIN_32:OUTPUT

BTN_L1 PIN_33:INPUT
```

После определения в .ve, при подготовке "LOGIC", выход "LED_D3" будет автоматически генерироваться в логическом модуле верхнего уровня, а затем канал "LED_D3" будет управляться кодом FPGA, который фактически является контактом "PIN_32". Тоже относится к "LED_D3". Линия кнопки "BTN_L1" "привязана" к пину "PIN_33" и работает только на ввод.

Если направление в .ve не определено, в FPGA автоматически генерируется направление типа "INOUT".

Ассоциация сигналов между MCU и FPGA

Формат определения:

```
function_name_FPGA signal_name_MCU
# комментарий
```

Где,

- function_name_FPGA - имя функции FPGA;
- signal_name_MCU - имя функции MCU.

Например, можно определить сигнал GPIO для FPGA - "GPIO4_3 iocvt_chn", или сигнал "tx" последовательного порта 1 для FPGA - "UART1_UARTTXD txd_chn", как показано ниже.

```
GPIO4_3       iocvt_chn
UART1_UARTTXD txd_chn
```

Примечание. Важно! Каждый раз, когда файл .ve изменяется, соответствующий исходный код .v будет автоматически генерироваться посредством подготовки "LOGIC". (Если файл .ve был изменен ранее, файл _templ.v будет создан при следующей подготовке "LOGIC" вместо прямой перезаписи файла .v. Вам нужно будет сравнить и объединить его самостоятельно).

Подготовка проекта FPGA

Просто создайте новый проект используя PlatformIO. Укажите имя проекта, устройство, SDK. После укажите директорию для проекта. Нажмите "Finish" - проект будет создан с указанными вами параметрами. Во вкладке слева появится содержимое проекта. Откройте файл "platformio.ini". В нём вы увидите некоторые настройки в секции [env:name_board], где name_board соответствует тому имени, которое указано в board.

На этом этапе в списке задач проекта ещё нет задачи "Prepare LOGIC". Добавте в .ini файл следующие две строки.

```
ip_name = analog_ip

logic_dir = logic
```

Сохраните настройки. Вы заметите, что после сохранения настроек, PlatformIO примется за работу и у вас появится новая задача "Prepare LOGIC". Это даст вам возможность создать проект logic "по умолчанию".

Теперь можно воспользоваться созданной задачей "Prepare LOGIC". В результате отработки этой задачи будет создан каталог "logic" с несколькими файлами проекта logic "по умолчанию" в нём.

О перезаписи имен файлов.

Имена файлов в “logic по умолчанию” основаны на элементах конфигурации в platformio.ini

```
[env:example_board]

board_logic.ve = example_board.ve

ip_name      = analog_ip

log_dir      = logic
```

Важно 1. Если вы хотите изменить имя файла, вы сначала изменяете имя в platformio.ini, затем вызываете задачу “Prepare LOGIC”, чтобы автоматически сгенерировать файлы с новыми именами.

Имена, которые можно изменить:

- board_logic.ve соответствует имени example_board.v в папке logic.
- ip_name соответствует имени analog_ip.v в папке logic.
- logic_dir соответствует имени папки logic.

В созданной папке обратите внимание на два файла .v: analog_ip.v и example_board.v. Эти два файла являются файлами исходного кода vlog. Где:

- analog_ip.v — пустой шаблон. Функции, которые будут реализованы пользователем, расширяются на этом пустом шаблоне;
- example_board.v — это исходный код v, автоматически преобразуемый во время подготовки logic на основе конфигурации контактов (PIN) в example_board.ve. Это также верхний (top) модуль проекта выше. Не изменяйте эту часть вручную.

Если в проекте есть изменения выводов, повторно подготовьте logic и экспортируйте их сюда после настройки в ve.

На этом этапе создается пустой проект. Это всего лишь пример. В реальных приложениях перед экспортом пустого проекта необходимо настроить другие элементы в platformio.ini и контакты, необходимые для example_board.ve. Это относится к настройке board_logic.device с чипами 32/48/64/100 контактов. Более подробную информацию о том, как настроить проект, можно получить на сайте документации PlatformIO. Ниже приводится полный пример файла настроек начального проекта.

```
# Секция для расширенных (пользовательских) настроек

[setup]
boards_dir = boards
board = agrv2k_407
framework = agrv_sdk
board_logic.ve = example_board.ve
ip_name = analog_ip
logic_dir = logic

# Секция верхнего уровня для настройки проекта

[env:AgRv] platform =
AgRV extends =
setup        # Common settings.
setup_logic  # Logic settings, include pin map, IP, etc.
setup_upload # Debug and upload settings. Replace with setup_upload_serial to use
             # serial upload. setup_monitor # Monitor settings. Replace with
             # setup_monitor_rtt to use Segger RTT.

setup_batch  # Batch binary settings.
```

Далее можно открыть созданный проект в Quartus. Проверьте, что выбран элемент EP4CE75F23C8 Cyclone IV E. В закладке Files, кроме уже упомянутых файлов, будет присутствовать ещё один файл alta_sim.v. Этот файл предоставляет функции системы чипа. Он похож на библиотеку функций, поэтому на него не нужно обращать внимание.

Пользователь реализует свои функции в analog_ip.v. Ещё раз посмотрите на проект.

1. analog_ip.v - вход в пользовательскую логику;
2. example_board.v - верхний модуль всей логики. Он свяжет модуль analog_ip с каждым модулем в atla_sim.
3. alta_sim.v - инкапсулированные модули, относящиеся к AG32.

Назовём пользовательскую логику “помигаем светодиодом”. Здесь я привожу модуль analog_ip так как он дан в примере.

```
module analog_ip (
  output    LED_D2,
  output    LED_D3,
  input sys_clock,
  input bus_clock,
  input resetn,
  input stop,
  ...

  );

  led led_output(
      .clk (sys_clock),
      .LED_CTL2 (LED_D2),
      .LED_CTL3 (LED_D3)
    );

endmodule

module led (
  input     clk,
  output    LED_CTL2,
  output    LED_CTL3

  );

  reg LED_2;

  reg LED_3;

  assign LED_CTL2 = LED_2;

  assign LED_CTL3 = LED_3;

  reg [31:0] clkcount;

  always @ (posedge clk)

  begin
  clkcount <= clkcount+1;
  case(clkcount[24:23])
    0:      LED_2 <= 0;
    1:      LED_2 <= 1;
    2:      LED_3 <= 0;
    3:      LED_3 <= 1;  endcase

  end

endmodule
```

LED_D2, LED_D3 это светодиоды, которыми будем моргать. Для выбора контактов, обратимся к рисунку 2

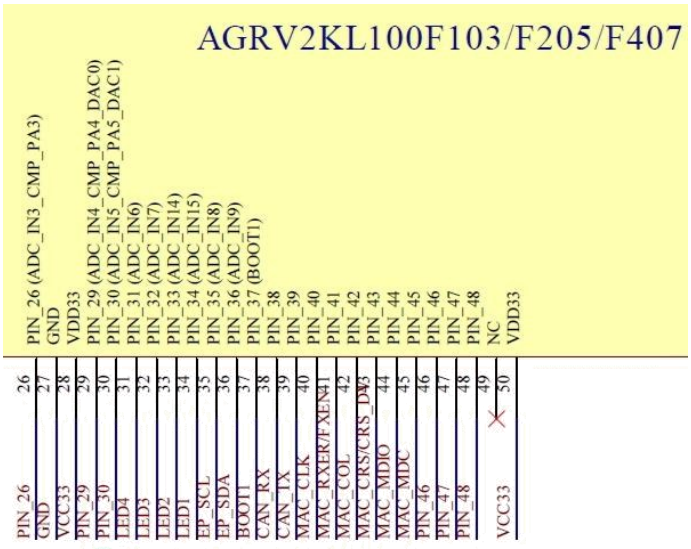


Рис. 2: Контакты LQFP100.

Соответствующие контакты, к которым они подключены заданы в файле example_board.ve

```
LED_D3 PIN_32:OUTPUT # LED2
LED_D2 PIN_31:OUTPUT # LED3
```

Опорные частоты sys_clock и bus_clock задаются там же

```
SYSCLK 100 # MHz
HSECLK 8   # MHz
```

Теперь перейдите в Quartus и выполните TCLScripts -> af_quartus.tcl. Эта команда будет генерировать файл .vo, в котором будет отражена логика проекта. Затем перейдите в “Supra”, чтобы продолжить создание .bin файла.

В инструменте “Supra” откройте проект (example/logic). Затем нажмите «Инструменты» -> «Компилировать» и нажмите «Выполнить» в правом нижнем углу всплывающего экрана. После успешной компиляции на экране появится подсказка.

```
Compile design example_board done with code 0
```

В директории logic будет находиться собранный файл example_board.bin. Получение этого файла являлось целью двух этапной компиляции при помощи “Quartus” и “Supra”. Теперь этим файлом можно “прожечь” микроконтроллер напрямую из Supra и наблюдать эманацию света из светодиодов LED.

Примечание. Если используется чистый FPGA, при определении и написании логической части не нужно учитывать сторону MCU. Вышеописанный процесс включает в себя два ключевых момента:

1. Именование пользовательских модулей: пользовательская логика, имя пользовательского файла должно совпадать с именем пользовательского модуля, которое представляет собой имя ip_name, установленное в platformio.ini.
2. Корреляция сигналов, определенная в ve: В AG32 MCU, FPGA и внешние контакты независимы друг от друга.
   a. Ввод/вывод, используемый MCU, в .ve, может быть связан с внешним выводом PIN_xx;
   b. Ввод/вывод, используемый FPGA, в .ve может быть связан с внешним выводом PIN_xx;
   c. Определенный сигнал MCU может быть напрямую связан с определенным сигналом FPGA в .ve;

Следовательно, .ve является критическим местом. После определения в .ve запуск “Prepare LOGIC” автоматически сгенерирует входные и выходные интерфейсы модуля верхнего уровня FPGA. Эти интерфейсы представляют собой пути прохождения сигналов, связанные с внешними выводами между FPGA и MCU.

В дальнейшем, данные действия будут рассмотрены более подробно на настоящем примере с функцией “Помигать светодиодом”.

Выводы

В статье рассмотрены особенности кристалла семейства AG32. Раскрыты технические характеристики и структурная схема кристалла, что даёт возможность быстро сориентироваться в применении семейства микроконтроллеров и определить сферу для их использования. Также описано необходимое программное обеспечение для работы с этими кристаллами.

Несомненным плюсом для небольших разработок является то, что данный кристалл содержит готовую “систему на кристалле” с процессором RISC-V и FPGA в “одном флаконе”. Обычно, имея только ‘FPGA приходится мириться с отсутствием процессора, что вынуждает решать проблемы при разработке либо только с использованием FPGA, либо придумывать различные нестандартные решения с использованием платы в формате Arduino. Генерировать “систему на кристалле”, потом писать код для неё ради пяти строк

полезного кода - это, возможно познавательно, но очень затратно.

Набор всех этих устройств в одном кристалле даёт широкие возможности для творчества при создании логических схем небольшого объёма. А при оптимальном распределении функций между MCU и FPGA можно создавать довольно интересные конструкции, которые при использовании только плат типа Ардуино либо вообще не возможно создать, либо это весьма затруднительно и накладно.

В заключение хочу отметить, что все необходимое для работы с отладочной платой, включая документацию, находится на российском GitVerse

https://gitverse.ru/AGM-micro

FPGA Design and Verification using opensource Workflows

- **Ajeetha Kumari Venkatesan**, Director of Verification ,*AsFigo Technologies, London, UK (ajeethak@asfigo.com)*
- **Deepa Palaniappan,** ASIC *DAV Engineer ,AsFigo Technologies* , Rapperswil, Switzerland (deepa@asfigo.com)
- **Hemamalini Sundaram,** Tech Lead, Verifworks Private Limited, Bangalore, KA – India, (hema@verifworks.com)
- **V P Sampath,** Founder, Bharath Semiconductor Society, Chennai, TN – India (ramsampath@bharathsemi.com)

Обсуждение и комментарии: link

Abstract

FPGAs are an excellent choice for AI workloads, offering quick prototyping of diverse training algorithms and customizable hardware acceleration for high-performance tasks. SystemVerilog, a widely used hardware description language, is central to FPGA-based digital design due to its expressiveness and capability to model complex systems. However, the increasing complexity of SystemVerilog RTL (Register-Transfer Level) designs introduces challenges such as coding errors, synthesis inefficiencies, and non-compliance with best practices. These issues can lead to functional mismatches, increased debugging time, and suboptimal hardware performance. Lint tools, which perform static analysis of code, are essential for ensuring high-quality SystemVerilog RTL design. They detect coding errors, enforce style guidelines, and highlight synthesis incompatibilities early in the design cycle, reducing costly iterations. With FPGAs demanding highly optimized RTL for efficient synthesis and performance, robust linting tools tailored to SystemVerilog are critical to streamline development, improve reliability, and ensure adherence to FPGA-specific constraints. This paper describes the role of lint tools in modern FPGA design workflows and explores their role in enhancing RTL quality for synthesis.

Introduction

FPGAs (Field-Programmable Gate Arrays) are reconfigurable semiconductor devices valued for parallel processing and hardware adaptability. Unlike fixed ASICs, FPGAs enable post-manufacturing programming for rapid prototyping and iterative design. They comprise programmable logic blocks, interconnects, and resources like DSP slices, memory, and sometimes embedded processors, balancing flexibility with performance. FPGAs excel in diverse fields, including AI, where they serve as accelerators by leveraging their customizable architecture for high-throughput and low-latency tasks. Advances in FPGA architecture and high-level synthesis drive their adoption in high-performance com-

puting and edge applications, bridging software flexibility with hardware efficiency.

The complexity of modern digital designs, especially for FPGA and ASIC workflows, necessitates sophisticated synthesis and linting solutions. While proprietary tools dominate the EDA industry, open-source alternatives are emerging as viable options due to their adaptability, cost efficiency, and integration capabilities. This paper explores the ecosystem of open-source synthesis tools and Python-based linting frameworks. These tools enable highly adaptable FPGA workflows, which benefit from the inherent reconfigurability and modularity of FPGAs.

Open-Source Synthesis for FPGAs

Yosys is an open-source framework for digital synthesis, widely used in FPGA design workflows. It supports a range of hardware description languages like Verilog and SystemVerilog (with limited support), enabling RTL-to-netlist transformations. Yosys is highly extensible, allowing developers to integrate custom passes and optimizations for FPGA-specific needs. Its compatibility with various FPGA architectures and integration with tools like nextpnr make it an asset for open-source FPGA synthesis.

Coding styles – Impact on FPGA resource usage

Coding styles in RTL design significantly impact FPGA resource utilization, timing performance, and overall synthesis efficiency. Poor coding practices, such as excessive use of nested conditional statements or poorly structured FSMs (Finite State Machines), can lead to inefficient mapping onto FPGA resources like LUTs, flip-flops, and DSP blocks. For example, asynchronous resets are harder to implement efficiently compared to synchronous resets due to FPGA-specific architectural constraints. Unintended latching behavior from incomplete case statements or combinational loops can increase resource usage and cause timing issues. Overusing high-level constructs such as *always_comb* or poorly constrained array indexing may lead to unnecessary logic replication. On the other hand, modular design, proper pipelining, and thoughtful use of FPGA primitives like BRAM or DSP slices can optimize performance and resource efficiency. Aligning coding styles with synthesis tool guidelines and FPGA architecture-specific recommendations ensures better utilization, higher performance, and more predictable results in FPGA design.

While we focus on open-source synthesis tools in this paper, most of the ideas/concepts apply equally to any commercial EDA tool tailored for FPGAs.

Vendor-Specific FPGA Architectures and Optimized HDL Coding Styles

FPGA architectures vary significantly across vendors and families, requiring HDL coding styles tailored to their specific features. For Xilinx FPGAs, Spartan devices target cost-sensitive applications with limited DSP and BRAM resources, favoring simple coding for area optimization. In contrast, Virtex and newer UltraScale families offer high DSP and memory density, suitable for pipelined architecture and heavily parallel designs. Altera (now Intel) FPGAs, such as Cyclone and Stratix families, emphasize high-speed transceivers and logic density, making them ideal for high-throughput designs when leveraging DSP and M20K memory blocks. Microsemi (formerly Actel) FPGAs, like the SmartFusion series, rely on flash-based architecture, favoring designs with low power consumption and secure operations, often using synchronous design principles for robustness. Lattice FPGAs, designed for low-power and small-form-factor applications, benefit from minimalistic and efficient HDL coding styles. Each vendor's unique archi-

tecture demands strategic resource utilization, pipelining, and careful inference of primitives to maximize performance.

Optimized HDL Coding Styles for Actel FPGAs

1. Avoiding Asynchronous Resets: Actel FPGAs favor synchronous resets for reliable timing closure and resource efficiency. Asynchronous resets may not map efficiently to the flash-based architecture.

2. Clock Gating Alternatives: Instead of manual clock gating, use enable signals or Actel's specific clock management features to reduce power consumption and maintain timing integrity.

3. Minimal Combinational Feedback: Flash-based FPGAs are sensitive to combinational loops, which can create unstable behavior. Properly structuring FSMs and avoiding unintended feedback loops is critical.

4. Pipeline Optimization: Efficient pipelining reduces logic depth, improving timing closure on Actel FPGAs, which have tighter timing margins compared to SRAM-based FPGAs.

5. Resource-Specific Mapping: Use Actel's hardened IP blocks, such as memory macros or secure cryptographic cores, instead of relying on generic constructs.

6. Efficient Memory Usage: Infer small memory arrays instead of large generic ones to match Actel's memory architecture, which prioritizes distributed over block RAM.

7. Tool-Aware Coding: Leverage Microchip's Libero SoC software toolset to adhere to Actel-specific synthesis guidelines, ensuring optimized resource utilization.

Build Your Own Linter (BYOL) for FPGA HDL Design Optimization

Building your own linter (BYOL) for FPGA HDL designs can be a highly effective way to enforce coding standards and optimize synthesis results for specific FPGA architectures, such as Actel or Xilinx. A custom linter can be tailored to catch common issues, improve resource usage, and ensure that design practices are aligned with the unique constraints of the target FPGA family.

1. Targeted Rule Sets: Custom linters can be created to enforce FPGA-specific best practices, such as avoiding asynchronous resets in Actel FPGAs, ensuring proper clock domain crossing, and optimizing memory usage. These rules can be fine-tuned based on the vendor (e.g., Xilinx, Intel, Lattice) and FPGA family (e.g., Spartan, Virtex, Stratix, IGLOO).

2. Integration with Synthesis Tools: A BYOL can integrate with existing synthesis flows (such as Vivado for Xilinx or Libero for Actel), providing early feedback during the RTL design phase. It can analyze the HDL code before synthesis, identifying issues that could lead to inefficient resource usage or timing violations.

3. Static Code Analysis: A custom linter can perform static code analysis to detect common problems such as unoptimized FSMs, unintentional combinational loops, improper resource inference, and inefficient use of memory blocks. It can flag these issues before they become synthesis problems, saving time and improving design quality.

4. Vendor-Specific Optimizations: By implementing linter rules that reflect the unique architecture of a target FPGA family, a BYOL ensures the design is optimized for the hardware. For example, rules for efficient use of DSP blocks, LUTs, and BRAMs in Xilinx FPGAs, or memory macros in Actel FPGAs, can be enforced.

5. Customizable and Scalable: A BYOL can be easily modified as new FPGA families or synthesis tools emerge. It can be expanded to include additional checks as design complexity grows, making it a scalable solution for teams working across various projects and FPGA families.

6. Improved Design Quality: A custom linter improves design quality by ensuring that coding styles adhere to vendor-specific guidelines, reducing the number of errors detected during post-synthesis verification and speeding up the overall design cycle.

yoYoLint: A BYOL Example Based on Slang/PySlang in Python

yoYoLint is a custom-built linter (BYOL) designed for SystemVerilog RTL design for Yosys, utilizing the Slang/PySlang framework in Python. By combining the power of Python with the Slang framework, yoYoLint provides a flexible and extensible solution for FPGA designers.

1. Integration with Slang/PySlang: yoYoLint leverages the Slang/PySlang framework, which provides a Python interface for parsing hardware description languages such as Verilog, SystemVerilog, and VHDL. This allows yoYoLint to analyze RTL code and detect potential issues at an early stage in the design process.

2. Custom Rule Creation: yoYoLint allows users to create custom rules tailored to specific FPGA families, such as Xilinx, Intel, Actel, or Lattice. For instance, it can flag improper use of asynchronous resets, which can lead to timing issues in certain FPGA architectures. It can also check for efficient utilization of FPGA-specific resources like DSP slices, LUTs, and BRAM.

4. Vendor-Specific Optimizations: yoYoLint can be extended to cater to the specific characteristics of various FPGA families. For example, it can check for efficient memory usage on Xilinx FPGAs by ensuring that BRAMs are used correctly or on Actel FPGAs by enforcing distributed memory over block RAM for small arrays. This helps align the design with the target FPGA's resource and performance constraints.

5. Python-Based Customization: Built in Python, yoYoLint offers great flexibility for customization. Developers can easily extend the linter to add new checks or modify existing ones. This makes it adaptable to evolving FPGA families, synthesis toolchain updates, and new design practices.

6. Real-Time Feedback: yoYoLint can be integrated into a continuous integration (CI) pipeline, enabling real-time linting feedback as designers write RTL code. This early detection of design flaws accelerates the development cycle and ensures adherence to coding standards throughout the project.

7. Improved Resource Utilization: By enforcing best practices specific to the FPGA architecture, yoYoLint helps optimize RTL code for better resource utilization, lower power consumption, and faster timing closure, ultimately leading

to more efficient FPGA designs.

Customization and Innovation

• Minimalistic Rules:

- Offers simple, extensible rules that can serve as templates, making it easy for users to develop and apply their own checks.

• Extensibility:

- Enables users to create proprietary linting rules for project-specific requirements.

- Allows customization of existing rule sets, adapting the framework to evolving design practices.

- Supports building new analysis applications, expanding the linting capabilities beyond typical checks.

C. Data Model Foundation

• Robust Data Model: Uses a data model inspired by Verilog's VPI object model, ensuring a consistent and structured representation of SystemVerilog code for efficient analysis.

• No Pattern-Matching Dependence: Unlike traditional linting approaches that rely on pattern matching tools like sed, grep, or awk, the framework avoids these methods, ensuring more reliable and maintainable linting processes.

- Built on PySlang, supporting SystemVerilog linting for synthesizable subsets.

- Includes constructs such as typedef, logic, enum, struct, package, and interface.

```
ef yYLChkNaming(lvCuScp):

 if (lvCuScp.kind.name == 'InterfaceDeclaration')
   lv_ident_name = str(lvCuScp.header.name)
   lv_exp_s = sv_suffix_d['intf']
   yYLChkNameSuffix ('NAME_INTF_SUFFIX', lv_ident
   FUNC_NO_2STATE_IN_INTF(lvCuScp)

ef INT_VAR_NYS(lvDecl):
 if (lvDecl.kind.name == 'DataDeclaration'):
   lvRID = 'INT_VAR_NYS'
   lv_dt_s = str(lvDecl.type).strip()
   if (lv_dt_s == 'int'):
     msg = 'SystemVerilog int data type is '
     msg += 'NYS - Not Yet Supported in Yosys'
     msg += str(lvDecl)
     yYLMsg (lvRID, msg)

ef LOGIC_PARAM_NYS(lvDecl):
 if (lvDecl.kind.name == 'ParameterDeclarationSta
   lvRID = 'LOGIC_PARAM_NYS'
   lv_dt_s = str(lvDecl.parameter.type).strip()
   if ('logic' in lv_dt_s):
     msg = 'SystemVerilog parameter of type logic
```

Case Studies:

Impact of the nested if else statement in FPGA LUT

FPGAs use Look-Up Tables (LUTs) to implement logic functions, and minimizing LUT usage is a key goal for FPGA designers. To achieve this, following efficient coding guidelines is crucial. In SystemVerilog, combining multiple conditional statements can lead to complex decision-making processes in the design. Nested `if-else` and cascaded `if-else` structures are often used to handle multiple conditions, but they can introduce inefficiencies. These nested structures create more complex conditional logic, which is mapped to a single LUT, potentially leading to higher resource utilization. Large logic blocks are resource-intensive, and there are often situations where certain conditions are irrelevant or "don't care." Designing with this in mind can reduce LUT usage and optimize the overall FPGA design.

```
always @(*) begin
    if (a > b) begin
        // Case when a is greater than b
        if (a == 4'b1111) begin
            result = 4'b1000; // Special case
if a is 1111
        end else begin
            result = 4'b0001; // Default case
when a > b but not 1111
        end
    end else if (a < b) begin
        // Case when a is less than b
        if (b == 4'b0000) begin
            result = 4'b1111; // Special case
if b is 0000
        end else begin
            result = 4'b0010; // Default case
when a < b
        end
    end else begin
        // Case when a is equal to b
        result = 4'b0100; // When a == b
    end
end
endmodule
```

Analyzing the specifications (and understanding input behavv), one may be able to optimize the same by using *case* instead of nested *if..else*.

```
always @(*) begin
    case ({a > b, a == 4'b1111, b ==
4'b0000})
        3'b100: result = 4'b1000;  // a > b
and a == 1111
        3'b101: result = 4'b0001;  // a > b
and a != 1111
        3'b010: result = 4'b1111;  // a < b
and b == 0000
        3'b011: result = 4'b0010;  // a < b
and b != 0000
        3'b001: result = 4'b0100;  // a == b
        default: result = 4'bxxxx; // Default
case for safety (should never reach here)
    endcase
end
```

This is a specific check that we are adding to yoYoLint (https://github.com/AsFigo/yoYoLint/issues/47)

Impact of resetting all flops in FPGA

To improve FPGA performance, reduce logic levels and alleviate routing congestion by resetting only flip-flops associated with data-valid signals. Sample the datapath only when the control path indicates the data is valid. This selective reset avoids unnecessary LUT usage and simplifies the design by focusing resets on relevant flip-flops.

One design style could be that designs reset only the flip-flops that hold valid signals. This controlled approach ensures that resets are applied only when necessary, reducing routing congestion and optimizing timing without un-

```
module dpath_cipher (
    input logic clk,                          //
Clock signal
    input logic rst,                          //
Reset signal
    input logic [127:0] data_in,              //
128-bit Data input
    output logic [127:0] data_out             //
128-bit Data output
);

    // Internal registers for holding data
    logic [127:0] reg_data;
    logic valid;                              //
Valid signal generated internally

    // logic to determine if the data is val-
id
    always_ff @(posedge clk or posedge rst)
begin
        if (rst) begin
            valid <= 1'b0;  // Reset valid
signal
        end else begin
            // Example: valid if data_in is
not zero, and its most significant byte is
within a range
            valid <= (data_in != 128'b0) &&
(data_in[127:120] >= 8'h10 && data_in
[127:120] <= 8'h1F);
        end
    end
```

```
    // Cipher data transformation logic
    always_ff @(posedge clk) begin
        if (valid) begin
            // Rotate the data, XOR with a
pattern, and add an offset
            reg_data <= {data_in[126:0], da-
ta_in[127]} ^
128'hA5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A5 +
128'h00000000000000000000000000000001;
        end else begin
            reg_data <= reg_data;  // Hold
the previous value if not valid
        end
    end

    // Output the processed data
    assign data_out = reg_data;

endmodule
```

Early Bug Detection are achieved a 30% reduction in LUT usage was achieved in a Cryptographic cipher IP.

Conclusion

In conclusion, adopting a "Build Your Own Linter" (BYOL) approach for FPGA target architectures offers significant advantages in optimizing design quality, resource utilization, and overall performance. By creating custom linting tools tailored to the specific requirements of FPGA designs, engineers can enforce coding standards that minimize resource-intensive operations, such as excessive LUT usage, unnecessary resets, or inefficient logic structures. This selective control allows for more efficient use of FPGA resources and ensures that the design is both scalable and optimized for timing.

Moreover, a BYOL approach enables fine-grained checks, such as validating the use of control signals or ensuring minimal routing congestion, which are critical in FPGA designs where timing closure and routing efficiency are paramount. By integrating a BYOL system, teams can reduce the likelihood of errors, improve code maintainability, and ensure that designs adhere to the best practices specific to FPGA architecture.

Ultimately, BYOL empowers FPGA designers to streamline their workflows, reduce time-to-market, and create high-performance, resource-efficient designs that can leverage the full potential of modern FPGA devices.

Блочно схемные элементы в DEEDS

Аверченко А.П.

E-mail: apaverchenko@omgtu.ru

Обсуждение и комментарии: link

Продолжаем изучать программный пакет Deeds (Digital Electronics Education and Design Suite) - комплекс для обучения и разработки цифровой электроники. Это свободно распространяемое программное обеспечение, для преподавания цифровой схемотехники. Программу можно свободно скачать с официального сайта www.digitalelectronicsdeeds.com.

Довольно часто удобно представлять схему или часть схемы в виде элемента типа «чёрный ящик» со своими входами/выходами и использовать его в проекте в виде отдельного компонента. При этом значительно улучшается восприятие всей схемы целиком и её «читаемость». Программное обеспечение Deeds позволяет создавать такие компоненты, состоящие из логических элементов, конечных автоматов и процессорных систем. В программной среде Deeds такие компоненты называются CBE (Circuit Block Element). В программе создаётся CBE компонент сохраняется на компьютере с расширением .cbe. После создания компонента его можно вынести на рабочую область проекта в виде схемного элемента столько раз, сколько это необходимо. Ограничение текущей версии Deeds (2.50.200) заключается в том, что компонент CBE не может в качестве своего составляющего элемента содержать другой компонент CBE. Сохраненный CBE компонент можно вставить в проект, воспользовавшись соответствующими пунктами меню. При этом происходит копирование всего содержимого компонента в проект, т.е. не возникает непосредственных межфайловых связей, а это значит, что если файл CBE компонента изменить, то это не приведёт к изменению в проекте, где его копия была загружена ранее. Файл CBE после вставления в проект может даже быть удалён, на проект где его копия находится это не повлияет. CBE компонент загруженный в проект изменить в проекте невозможно. Любой CBE компонент может быть извлечён из проекта и сохранен на диск в файл с расширением .cbe.

Создание CBE компонента происходит точно также как создание основной схемы, за исключением, что начинается с пункта меню File / New Block. При этом, для визуального отличия, при создание CBE компонента контур рабочей области окрашен в розоваты цвет. Для CBE входные и выходные пины

имеют своё отличное обозначение и их состав и номенклатура существенно меньше по сравнению с подобными пунктами меню при рисование схемы. Присваиваемое имя для пинов ограничено тремя символами и двумя символами для шин. Начертание и именование пинов при создание CBE компонента представлено на рисунке 1.

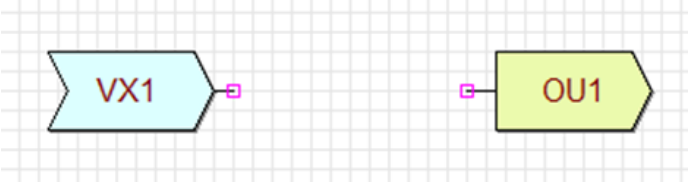


Рисунок 1 – графическое представление пинов CBE компонента

Редактирование символа CBE компонента: после создания компонента необходимо отредактировать его внешний вид, который будет использоваться при вставление его в проект. Редактированию подлежи размер, стороны расположения пинов и подпись самого компонента. Для открытия редактора графического представления компонента необходимо обратиться к пункту меню **View / Show Symbol Editor (F7)**. Откроется окно редактора, в котором и нужно будет настроить размер, стороны расположения пинов и подпись самого компонента.

После создания компонента и настройки внешнего вида его необходимо сохранить на диск с расширением .cbe.

Подключение CBE компонента к проекту происходит выбором пункта меню **Circuit / Components / Custom Components / Circuit Block Element (CBE)**, как это показано рисунке 2 или воспользоваться кнопкой на панели инструментов **Custom Components / Circuit Block Element (CBE)**, как это показано на рисунке 3 и рисунке 4.

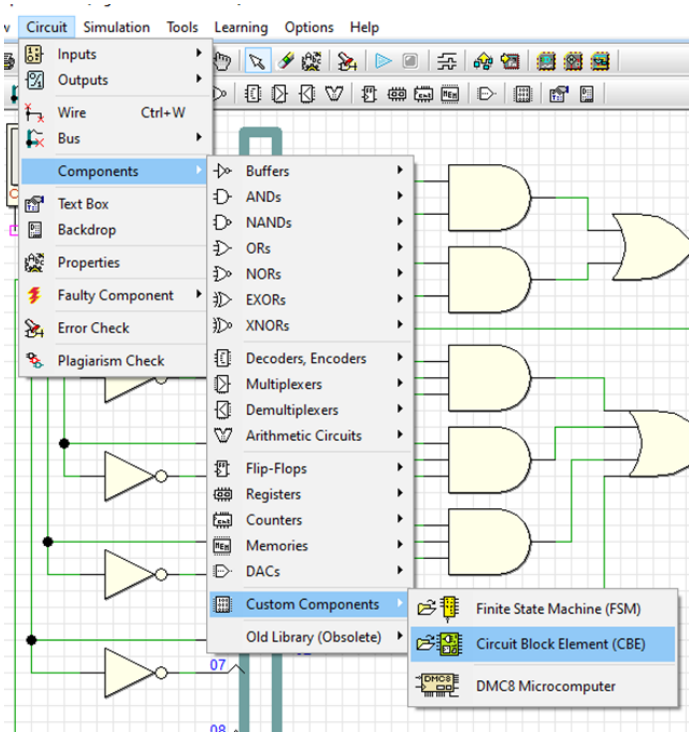


Рисунок 2 – меню подключения CBE компонент

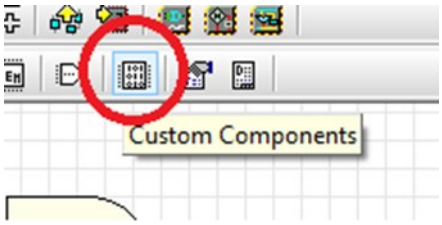


Рисунок 3 – кнопка на панели для подключения CBE компонента

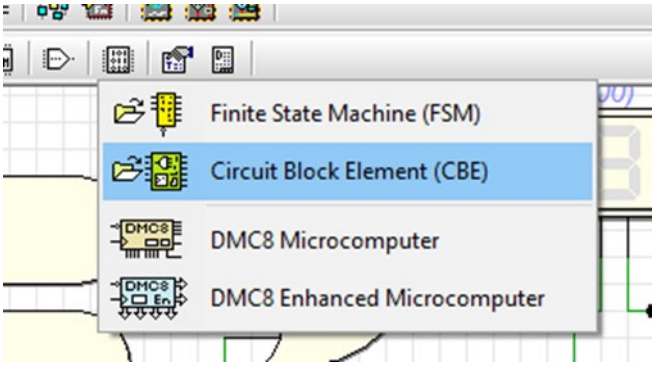


Рисунок 4 – кнопка Circuit Block Element (CBE)

Возможные варианты входных и выходных сигналов для компонента CBE представлены на рисунке 5. Диагностические точки «Test LED» единственная возможность отображать проходящие сигналы внутри компонента, выходы типа семисегментные индикаторы отсутствуют.

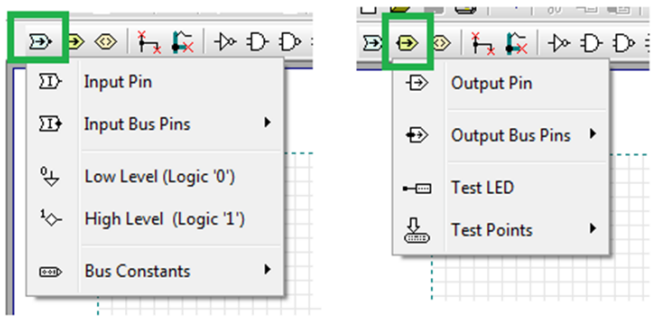


Рисунок 5 – входы и выходы CBE компонента

В режиме симуляции одинарным кликом на CBE компоненте открывается его содержимое и визуально отображается его работа, точно также как и работа всей схемы, показано на рисунке 6.

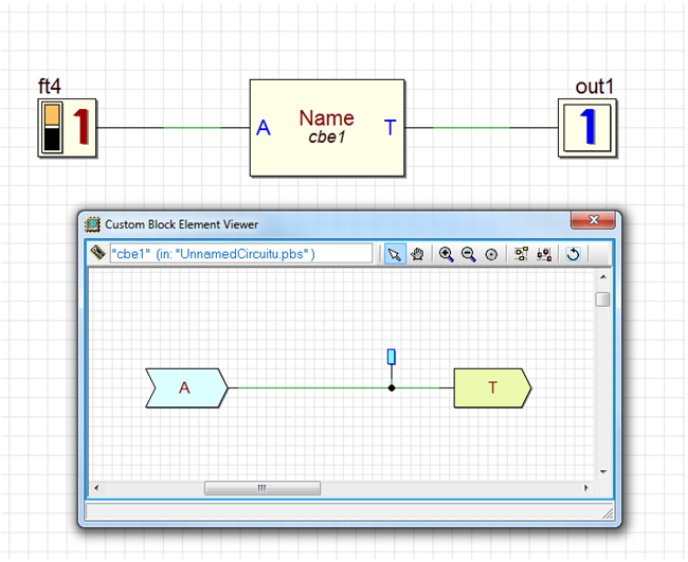


Рисунок 6 – симуляция работы CBE компонента

Коты приходят в Versal

Шанаева М., Попов М.

Обсуждение и комментарии: link

В прошлом номере мы вместе с вами, уважаемые дамы и господа, дружно радовались возможности за каких-то (не побоимся этого слова) 800 баксов, прикоснуться руками и головой к великолепному Versal [1].

Штош… надо признать, что прикосновения быстро наскучили, возникло ранее обещанное непреодолимое желание пойти дальше в освоении теперь уже предпоследних заокеанских технологий.

Тем, кто не знаком с содержанием прошлой серии [2], кратко напомним, что SoM V100 XCVE2302-SFVA784-1LP-E-S серии Versal AI Edge компании AMD. Главный чип объединяет в одном корпусе четыре процессора ARM Cortex-A72, два процессора Cortex-R5F, а также 34 модуля ускорения AI Engines-ML и 464 блока DSP. Модуль имеет два чипа флеш-памяти – QSPI емкостью 256 Мбит и одна микросхема eMMC емкостью 16 Гбайт. Процессорной части модуля доступно 53 линии ввода-вывода с уровнями 1.8 В, FPGA доступно 22 линии с уровнями 3.3 В, 54 линии 1.8 В, 30 линий 1.2 В и 8 пар высокоскоростных диффканалов GTY.

Целью же всего этого экшна будет как легкая попытка оценить возможность (и слегка показать методику) разворачивания на Petalinux пайтоновского кода средней простоты, так и ненавязчивая оценка производительности получившейся платформы в сравнении с настольным ПК и модулем Kria (в составе кита AMD KV260) в ходе выявления уникальных черт лиц котегов.

Нужно отметить, что в состав дистрибутива Petalinux, поставляемого в комплекте с VD100, входит Python 3.10. Состав пакетов не феноменален, но включает такие мастхэвы как *opencv*, *os* и *numpy*. К несчастью, величайшая *pip* в этот ансамбль не входит. Нас ждет либо перестроение образа (с приятной установкой актуальных инструментов AMD в десятки гигабайт), либо мы поставим *pip* ~~ручками~~ лапками, которые, как известно, у нас.

Последняя надежда установить *pip* разбилась о неумолимый экспешн. Поэтому нас все же ждет небольшое приключение минут на 20 (спойлер - нет, не 20).

Рассмотрим его по пунктам:

- установка Petalinux 2023.2 (строго так, и не спрашивайте, почему);
- скачивание заботливо подготовленных Alinx компактных архивов проекта (согласно методике [3]) объемом всего лишь около 85 Гб (будучи официальными партнером AMD, Alinx разместила эти архивы на серверах AMD, незримо указывая на VPN российским энтузиастам версального дела);
- добавление желанного *pip* путем вставки следующей строки в файл *<project_dir>/build/conf/local./conf* в папке проекта:

```
IMAGE_INSTALL : append = “ python3-pip”
```

- добавление не столь желанного, но очень нужного *cmake* следующей строкой в файле *<project_dir>/project-spec/meta-user/conf/user-rootfsconfig:*

```
CONFIG_cmake
```

включение добавленного *cmake* в образ в конфиге RootFS с помощью команды:

```
petalinux-config -c rootfs
```

- добавление *gcc* (в комментариях не нуждается и нашего желания не спрашивает) путем выбора пакета build-essentials в конфигурации RootFS;
- построение Petalinux традиционной командой petalinux-build (это может *take a while* часика на 3..4), а если в финале звучит ругань про сбой компиляции *qtwebkit*, то проверяем наличие пакетов *chrpath*, *cpio*, *diffstat*, *file*, *g++-multilib*, *gawk*, *gcc-multilib*, *git-core*, *locales*, *openssh-client*, *python*, *python3*, *socat*, *texinfo*, *tmux*, *unzip*, *wget*, доставляем нужные;
- создание файла *boot.bin* командой

```
petalinux-package --boot --u-boot --force;
```

- подготовка загрузочной MicroSD (довольно тривиально, гайд [4]).

После успешной загрузки Petalinux логинимся как petalinux (тут же от нас потребуют задать пароль), ставим пайтоновские пакеты *dlib* и *scikit-image* с помощью предусмотрительно подселенной нами в образ ранее *pip*:

```
sudo pip3 install dlib scikit-image
```

Сильно при этом не пугаемся, так как установка-билд *dlib* может занимать пару-тройку часов. Если это пугающе скучно, то можно запускать *pip* с ключом *-v*.

По дефолту ставится пакет *numpy 2.x*, что не нравится *OpenCV*, так что дауншифтим до 1.x:

```
sudo pip3 install --force-reinstall -v
"numpy==1.26.4"
```

Теперь находим подходящего котика и запускаем наш простенький софт, суть которого в нахождении 16 лендмарков, характеризующих уникальность каждого котолица [5]. Это делается с помощью библиотеки *dlib* с нейронкой, обученной на триллионах котиков нашей и соседних Галактик. Для визуализации соединим лендмарки отрезками (рис. 1).

Рис. 1 – Честно распознанный нейронкой ~~соавтор~~ кот Бакс

Ну и в порядке легкого челленджа для Versal - сравнение среднего времени анализа одного фото в сете из 100 котолиц [6]. *"Get ready to ruuuuuuuuuuuumble!"* © Michael Buffer, да-да, именно Buffer))

Итак, результат (немного удивительный) на табло, а мы с вами: 1) научились строить Petalinux для AMD Versal; 2) научились подкладывать в светлый petaобраз нужные софтинки и пакетики; 3) сохранили остатки психического здоровья в ходе работы с Petalinux (впрочем, не факт, что это касается и авторов).

P.S. Конечно же, искушенный версальской роскошью читатель обязан спросить: "минуточку, а как же AI Engines?!". И вопрос совершенно справедлив, но ответ мы рассчитываем дать в следующей части нашего ~~Марлезонского~~ Версальского балета.

Литература

1. AMD Versal AI Edge Series Product Selection Guide. XMP464 (V1.10). AMD, 2024. 7 p.

2. *Попов М. А., Романов А. Ю.* Versal… как много в этом слове! // FPGA – Systems magazine. 2024. № 2. С. 5–7.

3. VD100_2023.2. — About PETALINUX // GitHub : [Электронная платформа]. URL: https://github.com/alinxalinx/VD100_2023.2/blob/master/Demo/course_s2/documentations/EN/2_About_PETALINUX.md (дата обращения: 15.11.2024).

4. VD100_2023.2. — Quickly start Linux-make an SD card to start the development board Linux system // GitHub : [Электронная платформа]. URL: https://github.com/alinxalinx/VD100_2023.2/blob/ master/ Demo/ course_s2/documentations/EN/1_Quickly_start_Linux-Make_an_SD_card_to_start_the_development_board_Linux_system.md (дата обращения: 15.11.2024).

5. *Попова А. М., Попова В. М., Попов М. А.* Применение искусственного интеллекта для поиска и идентификации домашних животных // Труды 66-й Всероссийской научной конференции МФТИ. Радиотехника и компьютерные технологии. — М: Физматкнига, 2024. С. 113–115.

6. 5.7K Cat face labeled image dataset. Датасет кошачьих лиц [Электронный ресурс]. URL: https://images.cv/dataset/cat-face-image-classification-dataset (дата обращения: 15.11.2024).

Таблица 1 – Среднее время анализа одной фотографии на различных аппаратных платформах

Аппаратная платформа	AMD KV260	Alinx VD100	Intel i7-9700F @ 3 GHz
Время, с	1.05	1.79	0.24

Реверсивный счётчик с семисегментным индикатором на ATF22V10

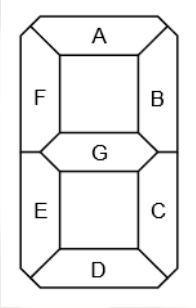
Харабадзе Д.Э.

Обсуждение и комментарии:

Удобным устройством отображения цифровой информации является 7-сегментный индикатор [1]. Одним из достоинств 7-сегментного индикатора является простота схемы управления, что позволяет применять для управления простые микросхемы, такие как ATF22V10 [2,3].

Светодиодный семисегментный индикатор представляет собой 7 прозрачных сегментов, каждый из которых может подсвечиваться своим светодиодом. Расположены сегменты таким образом, что подсвечивание различных сегментов приводит к отображению цифр от 0 до 9. Стандартная нумерация сегментов приведена на рисунке.

Для отображения различных цифр необходимо подсвечивать сегменты в соответствии с таблицей (1- светящийся сегмент, 0 — не светящийся):



		Разряды индикатора						
		A	B	C	D	E	F	G
Цифры	0	1	1	1	1	1	1	0
	1	0	1	1	0	0	0	0
	2	1	1	0	1	1	0	1
	3	1	1	1	1	0	0	1
	4	0	1	1	0	0	1	1
	5	1	0	1	1	0	1	1
	6	1	0	1	1	1	1	1
	7	1	1	1	0	0	0	0
	8	1	1	1	1	1	1	1
	9	1	1	1	1	0	1	1

Обычно для счёта импульсов и отображения на семисегментном индикаторе используют двоично-десятичный счётчик [4] и преобразователь двоичного кода в семисегментный [5]. Но применяя микросхему ATF22V10, можно реализовать функционал реверсивного счётчика с преобразователем в семисегментный код на одной микросхеме, производя счёт сразу в семисегментном коде.

Для отображения будем использовать индикатор GNS-56118S-21. У этого индикатора аноды всех светодиодов соединены вместе, поэтому их надо подключить к напряжению питания, а для того, чтобы зажечь нужный разряд, необходимо через резистор подключить катод соответствующего светодиода к общему проводу (То есть, подать «0»).

Соберём схему, изображённую на рисунке 1.

К первому выводу микросхемы подключена кнопка счёта, ко второму подключен переключатель выбора направления счёта. При наличии на второй ножке напряжения питания, счёт будет осуществляться в сторону увеличения. При наличии на второй ножке нулевого напряжения, счёт будет осуществляться в сторону уменьшения. Выводы 16-22 микросхемы через резисторы сопротивлением 1 кОм соединены с катодами светодиодов соответствующих сегментов.

Для простоты вывод сброса не используется. Сброс будет осуществляться при первом нажатии на кнопку CLK. В случае отображения непредусмотренной информации, счётчик сбросится в состояние «0».

Микросхема прошивается в соответствии с описанием на языке CUPL [6]

```
Name     Segment ;
PartNo   00 ;
Date     21.07.2024 ;
Revision 01 ;
Designer David ;
Company  FPGA-Systems.ru ;
Assembly None ;
Location  ;
Device   G22V10 ;

/* *************** INPUT PINS
*********************/
PIN 1 = CLK ;
PIN 2 = FORW ;

/* *************** OUTPUT PINS
*********************/
PIN 22 = !SA ;
PIN 21 = !SB ;
PIN 20 = !SC ;
PIN 19 = !SD ;
PIN 18 = !SE ;
PIN 17 = !SF ;
PIN 16 = !SG ;

/* *************** INTERNAL NODES
****************/
PIN 14 = LRESET ;

/* ----------------- Decode 7-seg -----------
--- */
S0 =  SA &  SB &  SC &  SD &  SE &  SF & !SG;
S1 = !SA &  SB &  SC & !SD & !SE & !SF & !SG;
S2 =  SA &  SB & !SC &  SD &  SE & !SF &  SG;
S3 =  SA &  SB &  SC &  SD & !SE & !SF &  SG;
S4 = !SA &  SB &  SC & !SD & !SE &  SF &  SG;

S5 =  SA & !SB &  SC &  SD & !SE &  SF &  SG;
S6 =  SA & !SB &  SC &  SD &  SE &  SF &  SG;
S7 =  SA &  SB &  SC & !SD & !SE & !SF & !SG;
S8 =  SA &  SB &  SC &  SD &  SE &  SF &  SG;
S9 =  SA &  SB &  SC &  SD & !SE &  SF &  SG;

/* ----------------- Next value -------------
--- */
LRESET = (S0 # S2 # S3 # S4 # S5 # S6 # S7 #
S8) # (S1 & FORW) # (S9 & !FORW);
P0= !LRESET ;
P1=S0 & FORW # S2 & !FORW;
P2=S1 & FORW # S3 & !FORW;
P3=S2 & FORW # S4 & !FORW;
P4=S3 & FORW # S5 & !FORW;
P5=S4 & FORW # S6 & !FORW;
P6=S5 & FORW # S7 & !FORW;
P7=S6 & FORW # S8 & !FORW;
P8=S7 & FORW # S9 & !FORW;
P9=S8 & FORW # S0 & !FORW;
```

```
/* --------------- Encode 7-seg -------------
--- */
SA.D = P0 #      P2 # P3 #      P5 # P6 # P7
# P8 # P9 ;
SB.D = P0 # P1 # P2 # P3 # P4 #           P7
# P8 # P9 ;
SC.D = P0 # P1 #      P3 # P4 # P5 # P6 # P7
# P8 # P9 ;
SD.D = P0 #      P2 # P3 #      P5 # P6
#      P8 # P9 ;
SE.D = P0 #      P2 #                P6
#      P8      ;
SF.D = P0 #                P4 # P5 # P6
#      P8 # P9 ;
SG.D =           P2 # P3 # P4 # P5 # P6
#      P8 # P9 ;

/* --------------- Set and Reset ------------
--- */
SA.AR = 'B'0;
SB.AR = 'B'0;
SC.AR = 'B'0;
SD.AR = 'B'0;
SE.AR = 'B'0;
SF.AR = 'B'0;
SG.AR = 'B'0;

SA.SP = 'B'0;
SB.SP = 'B'0;
SC.SP = 'B'0;
SD.SP = 'B'0;
SE.SP = 'B'0;
SF.SP = 'B'0;
SG.SP = 'B'0;
```

Приведённое описание расшифровывает семисегментный код в позиционный (где число задаётся положением единичного бита), после чего, в зависимости от состояния переключателя DIR увеличивает или уменьшает его. После этого преобразует сигнал в семисегментный код и при нажатии на кнопку CLK запоминает в выходные регистры новое значение.

Для того, чтобы обеспечить сброс в «0» при неправильном состоянии выходных ножек, добавлена линия сброса LRESET (14-я ножка). При счёте в сторону увеличения на линии LRESET появляется напряжение питания при отображении цифр 0..8, значит, во всех других случаях, при поступлении тактового импульса, на индикаторе появляется ноль. При счёте в сторону уменьшения на линии LRESET оказывается напряжение питания при отображении цифр 0,2..9, поэтому во всех остальных случаях на индикаторе появится наль.

Видно, что к ножкам 16-22 подключены инверторы, так как соответствующий сегмент индикатора загорается при появлении низкого уровня на ножке. При использовании семисегментного индикатора с общим катодом от инверторов следует отказаться.

[1] https://en.wikipedia.org/wiki/Seven-segment_display

[2] https://www.microchip.com/en-us/product/atf22v10c

[3] https://en.wikipedia.org/wiki/GAL22V10

[4] https://okbexiton.ru/pdf/mc1564ie6.pdf

[5] https://okbexiton.ru/pdf/mc1564id23.pdf

[6] https://ww1.microchip.com/downloads/en/DeviceDoc/doc0737.pdf

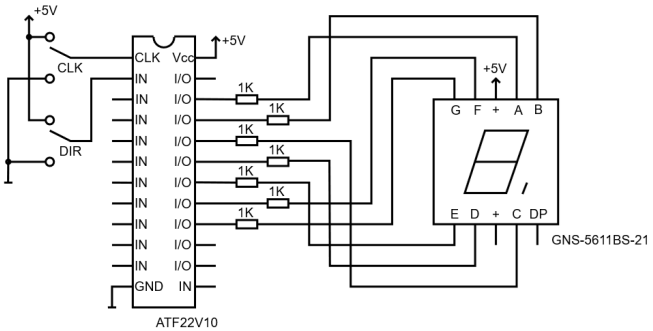


Рисунок 1. Схема подключения

Стереокамера машинного зрения с поддержкой ИИ на базе FPGA и Arduino Portenta H7

Буренков Сергей Алексеевич
E-mail: burenkov@list.ru
Telegram: @SergeyBurenkov

Обсуждение и комментарии: link

Введение

В статье рассмотрены детали реализации модуля стереокамеры на базе двух монохромных сенсоров MT9V034 с глобальным затвором. Для сшивания картинки и управления матрицами используется FPGA Gowin. Модуль подключается в качестве «шилда» к промышленной отладочной плате Arduino Portenta H7. Комбинированный видеопоток обрабатывается библиотекой машинного зрения OpenMV. Отладка проекта ведется в специализированной IDE от OpenMV на MicroPython, что позволяет быстро прототипировать устройства с использованием стерео-зрения. После отладки камера работает автономно, весь код исполняется микроконтроллером на Arduino. В библиотеке OpenMV реализовано большое количество функций обработки изображений, от базовых преобразований и фильтров, до машинного обучения. Точная синхронизация сенсоров с помощью FPGA позволяет выполнять поиск объектов на стереопаре, сопоставлять их и рассчитывать расстояние до этих объектов. Так же в библиотеке реализованы функции построения карты глубин, что позволяет использовать разработанную камеру для реализации алгоритмов автономной навигации.

Характеристики камеры

- Сенсоры: MT9V034C12STM
- Максимальное выходное разрешение: 1504x480
- Глубина цвета: до 10бит, монохромный.
- Частота кадров на максимальном разрешении: 60FPS
- FPGA : GW2AR-LV18QN88C8/I7
- Logic units(LUT4):20736
- Block SRAM (B-SRAM)(bits):828K
- 32bits SDR SDRAM 64M bits
- Numbers of 18x18 Multiplier 48
- Numbers of PLLs 2
- Конфигурационная память W25Q64JVSSIQ 64Мб
- Слот SD карты

Этапы реализации проекта

Прототип

Прототип был собран на базе проекта Robot Navigation using Stereo Vision , отладочной платы Arrow Deca MAX10(2) и Arduino PRO Portenta H7 Carrier Board (3)

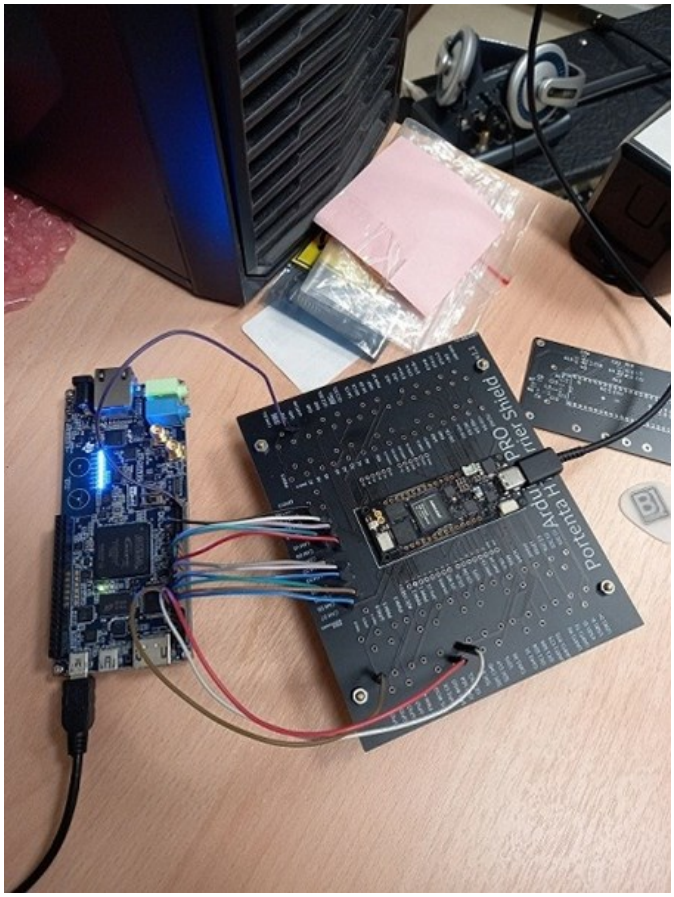


Рисунок 1. Внешний вид первого прототипа

На первом этапе прототипирования были отработаны способы взаимодействия с Portenta H7 и OpenMV. Нужно было чтобы IDE смогла при подключении идентифицировать плату Deca как сенсор изображения, в противном случае невозможно запустить скрипт MicroPython. OpenMV из коробки поддерживает Arduino Portenta H7, к которой подключается шилд с сенсором HM-01B0, следовательно необходимо было использовать I2C slave IP, который бы по запросу от Arduino имитировал ответ от моей платы аналогичный ответу HM-01B0. Поскольку проект OpenMV открытый (4), не составило труда найти адрес и номера регистров, к которым обращается процессор для идентификации сенсора, оказалось достаточно лишь ответить на запрос ID чипа по нулевому адресу.

I2C Slave To Avalon-MM Master Bridge Intel FPGA IP конвертирует запрос от шины I2C в протокол шины Avalon MM, а простейший самописный модуль HM-01B0_I2C_Stub подставляет нужные данные по запросу от Avalon Master.

Некоторая сложность возникла с выяснением того в какой момент производится опрос сенсоров. Оказалось, что он делается сразу при включении Arduino а не по нажатию кнопки «Connect» в IDE. При одновременной подаче питания на мой прототип и на Arduino прошивка OpenMV стартовала быстрее чем проект на плате Deca MAX10 и не успевала обнаружить сенсор, таким образом скрипт запустить не удавалось. В качестве временного решения я просто нажимал кнопку сброса Arduino после загрузки проекта FPGA на плате Deca MAX10.

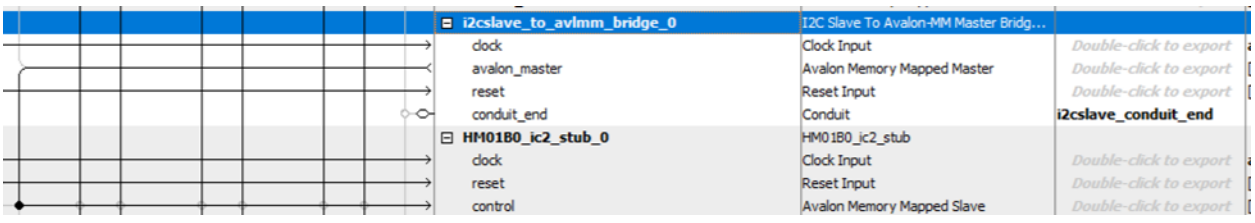


Рисунок 2. Intel Platform Designer. Комбинация из двух IP модулей для ответа на I2C запросы

После того как все эти моменты были учтены, удалось запустить получить картинку в среде OpenMV IDE.

Передаваемое изображение в виде монохромных вертикальных полос разной яркости было сгенерировано внутри FPGA с помощью Test Pattern Generator II Intel FPGA IP и Clocked Video Output II Intel FPGA IP

Следующим этапом нужно было получить изображение с сенсоров, для этого была заказана плата из проекта (1), на которую было смонтировано два видео сенсора MT9V034.

Кроме этого, необходимо было добавить в OpenMV драйвер для разрабатываемого модуля, так как драйвер для HM-01B0 не поддерживал нужного разрешения. Для этого был создан форк прошивки OpenMV, в который я добавил минимально необходимый драйвер, и указал его как поддерживаемый для платы Arduino Portenta H7. Инструкция по сборке проекта и прошивки платы есть в оригинальном репозитории, поэтому я опущу описание этих шагов. Версия библиотеки с моим драйвером доступна га гитхабе https://github.com/BurenkovS/openmv/.

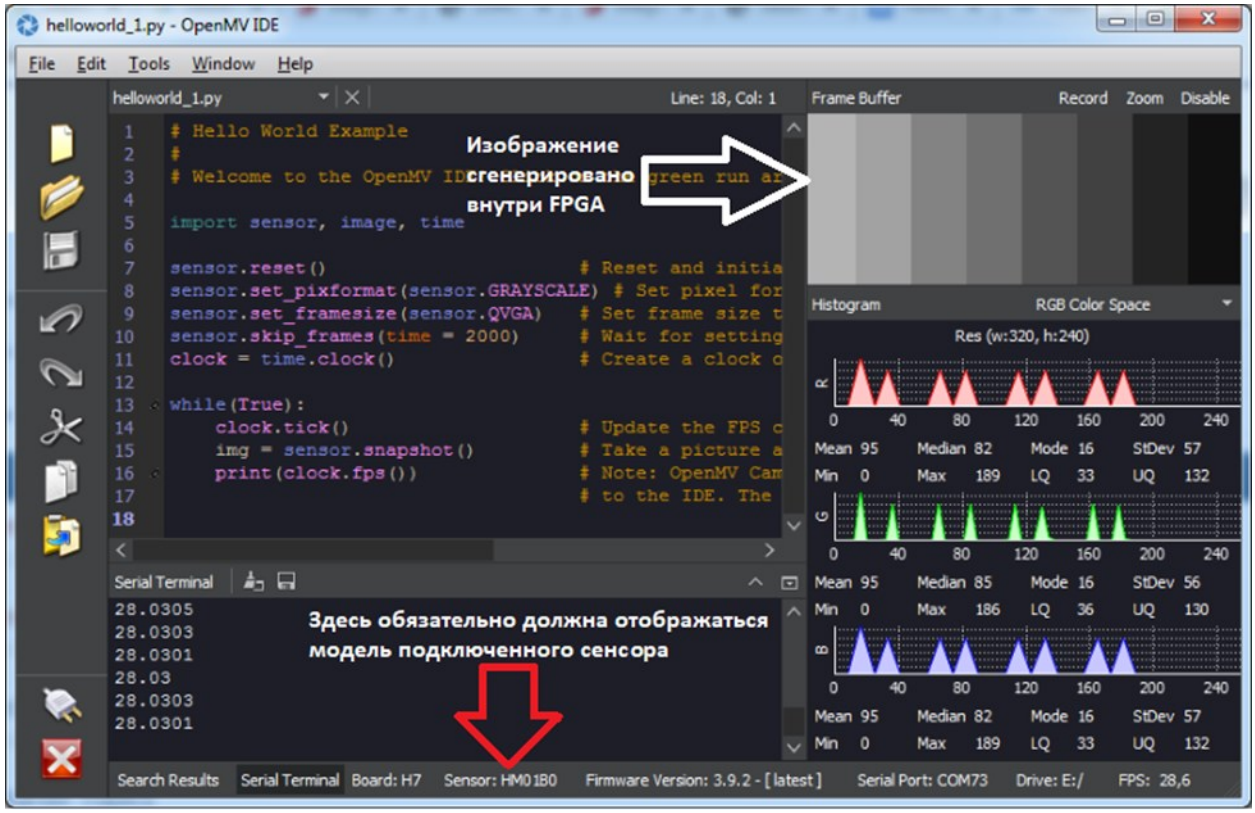


Рисунок 3. Скриншот рабочего окна OpenMV IDE

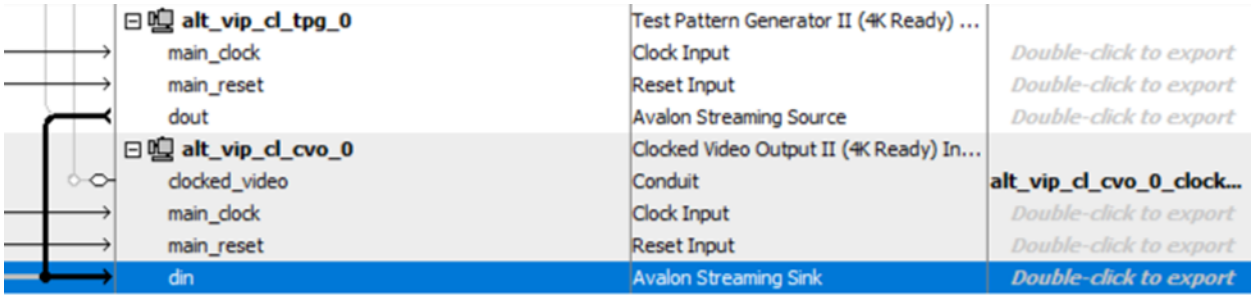


Рисунок 4. Intel Platform Designer. Блоки для генерации изображения

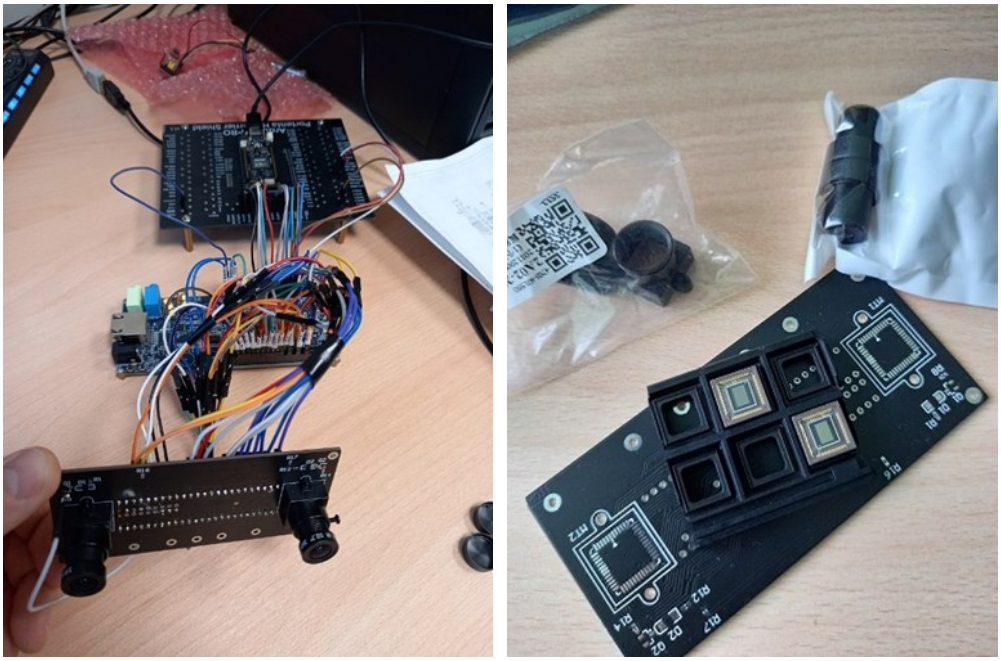


Рисунок 5. Прототип с видеосенсорами

Видео с сенсоров было обработано с помощью модулей из пакета «Intel Video and Image Processing Suite» (VIP)(5), который содержит в себе необходимые модули для захвата, обработки и вывода видеопотока. На данном этапе прототипирования сенсоры были инициализированы по умолчанию - в режиме master mode с автоматическим контролем экспозиции. Это позволило не разрабатывать алгоритм синхронного управления сенсорами на данном этапе, а использование пакета VIP Suite облегчило задачу синхронизации и склейки изображений двух источников. В конце концов картинка со стереопары была получена. Из-за навесного монтажа и огромного количества проводов картинка получилась шумной, работа модуля была нестабильна из-за помех или обрыва контакта одного из проводов. Самой большой проблемой была несинхронная работа сенсоров, что напрочь убивало идею получения идеально синхронизированной стереопары с помощью FPGA.

Разработка собственной платы

После выявления всех недостатков разработанного прототипа было принято решение проектировать собственный модуль. Кроме пары сенсоров и FPGA на модуле заложены слот SD карты, который подключается к SDIO интерфейсу Portenta H7, конфигурационная память для FPGA и разъем программирования JTAG. Для отладки на плате предусмотрено два светодиода и пара контактных площадок, которые я использовал для подключения UART-USB преобразователя для вывода отладочных сообщений. Разработка схемы и платы была заказана у опытного специалиста, и после нескольких недель обсуждения, месяца разработки и месяца изготовления был получен собственный модуль стереокамеры. Готовый модуль представлен на изображении

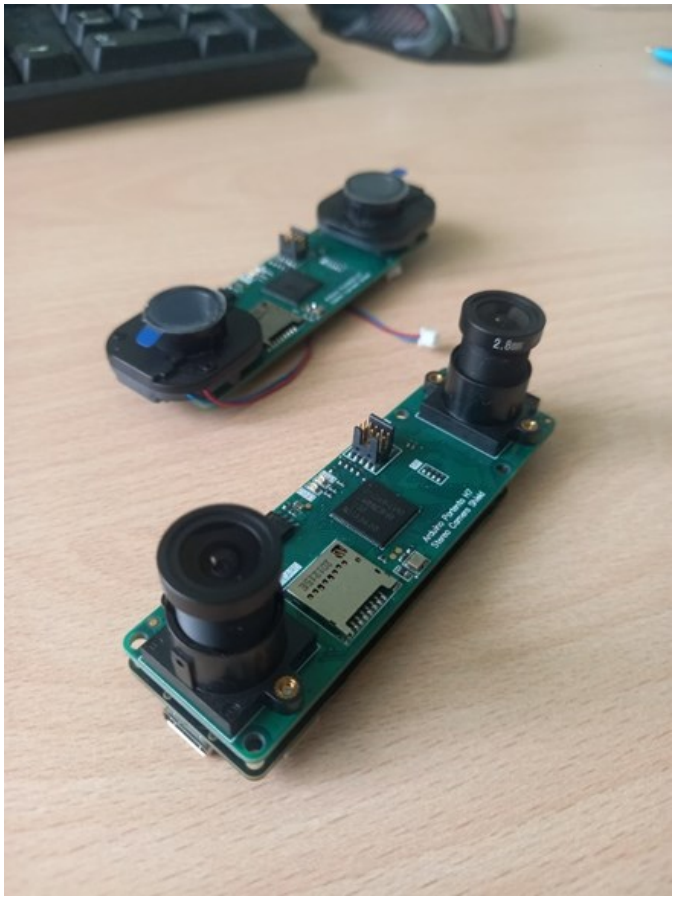


Рисунок 6. Первые экземпляры модуля

Разработка прошивки FPGA

Структурная схема проекта представлена на рисунке 7.

Основные блоки и их назначение:

Софт-процессор RISC-V PicoRV32. Процессор осуществляет начальную инициализацию сенсоров и модулей камеры, принимает команды по I2C от платы Arduino и перенастраивает параметры в зависимости от принятых команд. В настройках процессора включены модули I2C Master, OpenWB(мастер шины Wishbone) и UART. Процессор использует ончип память для кода и данных. Порт OpenWB подключен к блокам регистров а так же к модулю I2C Slave, который принимает запросы от Arduino. Адресное пространство со стороны Arduino поделено на 3 части, первая часть – это

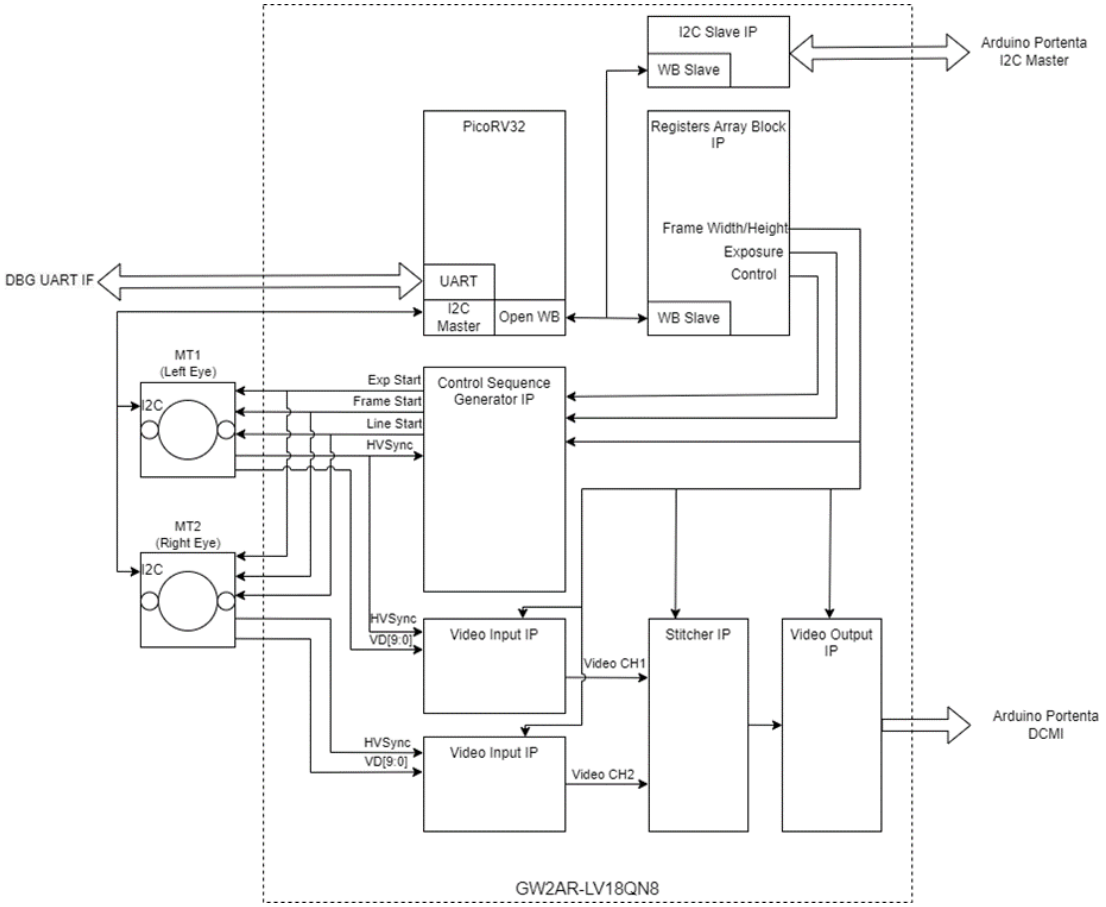


Рисунок 7. Структурная схема проекта

область общих параметров камеры. Втрое и третье адресное пространство напрямую отображается на оба сенсора соответственно, при записи по этим адресам I2C Master IP осуществляет запись по соответствующему адресу сенсора. Таким образом сохраняется возможность конфигурации любых параметров сенсора напрямую из Arduino. Кроме того, процессор содержит модуль UART, который используется для отправки отладочных данных через внешний UART-USB преобразователь.

Registers Array Block IP. Это простой модуль регистров, в который по шине Wishbone процессор записывает параметры системы. Содержит 4 регистра – разрешение по вертикали, разрешение по горизонтали, время экспозиции и контрольный регистр. Контрольный регистр управляет включением генерации управляющих сигналов для сенсоров в блоке Control Sequence Generator IP.

Control Sequence Generator IP(CSG). Формирует сигналы для управления сенсорами в режиме Slave Sequential Mode. На рисунках ниже представлена схема соединения управлявшего контроллера и сенсора(6), а также диаграмма управляющих сигналов в этом режиме(7).

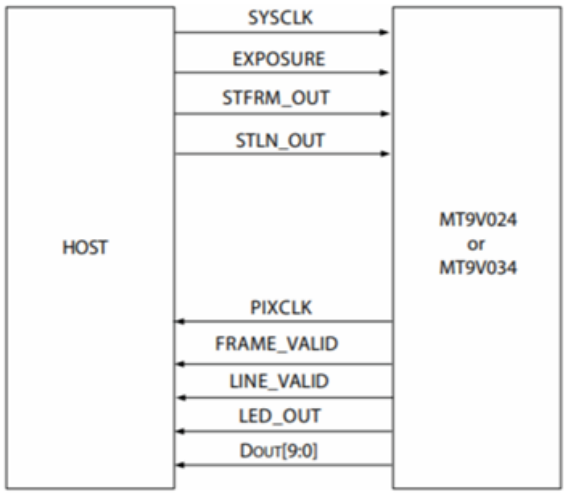


Рисунок 8. Схема соединения контроллера и сенсора

Генератор формирует сигналы EXPOSURE, STFRM_OUT и STLN_OUT одновременно для обоих сенсоров. По импульсу EXPOSURE начинается экспозиция кадра, по сигналу STFRM_OUT экспозиция останавливается, заряд из светочувствительных элементов переносится в область памяти для считывания и начинается процесс считывания кадра. Далее генератор начинает выдавать импульсы на выход STLN_OUT для старта считывания очередной строки изображения. В начале кадра считываются так называемся “область гашения” или vertical blanking, в это время выходы FRAME_VALID и

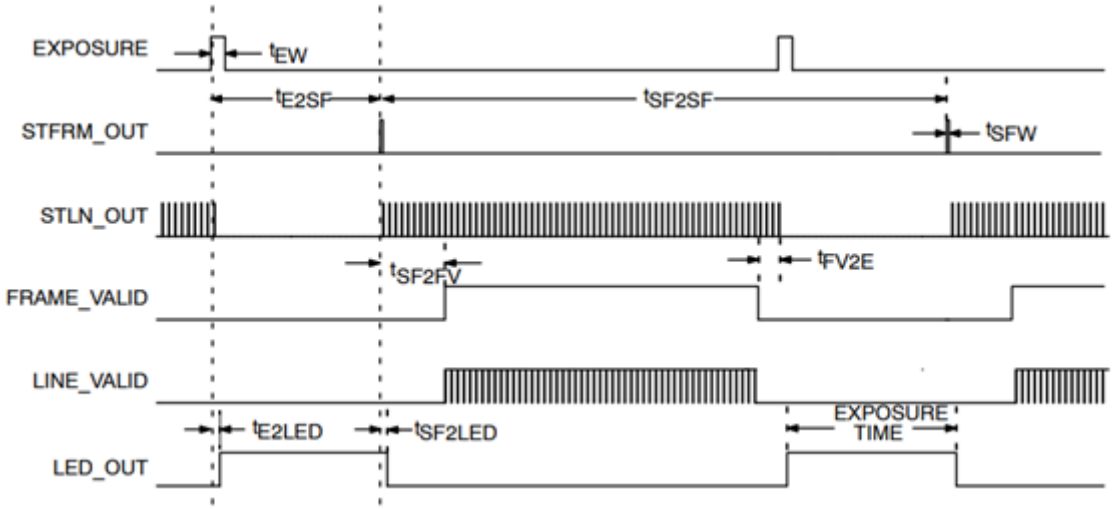


Рисунок 9. Диаграмма управляющих сигналов

LINE_VALID не активны, однако необходимо формировать STLN_OUT так чтобы вся строка изображения могла быть полностью вычитана из сенсора, т.е. учитывать заданную ширину строки. Далее генератор продолжает выдавать импульсы STLN_OUT, на выходах FRAME_VALID и LINE_VALID появляются активные сигналы, маркирующие активное изображение на кадре. После полного считывания кадра необходимо выдать еще два импульса 2 затем не ранее чем через 4 такта системной частоты сенсора можно вновь выставлять сигнал старта экспозиции. Несоблюдение последовательности или времени выдачи управляющих сигналов приводит к тому, что сенсор начинает выдавать данные в хаотичном порядке, пришлось потратить значительное время, чтобы привести автомат генератора к стабильной работе при любом заданном разрешении сенсоров. При нормальной работе сигналы FRAME_VALID и LINE_VALID с обоих сенсоров идентичны, поэтому на Control Sequence Generator заведены сигналы лишь с одного сенсора.

Video Input IP. Захватывает данные активного изображения от одного сенсора на его выходной тактовой частоте 27МГц, записывает данные в dual clock FIFO и выдает наружу на системной частоте 100МГц, дополняя видео поток одним дополнительным сигналом – маркером первого пикселя в кадре. Два модуля Video Input принимают на вход данные по независимым частотам от сенсоров и формируют выход в одном тактовом домене.

Stitcher IP. Формирует кадр удвоенной ширины из двух параллельных входных потоков. Состоит из управляющего автомата, мультиплексора и трех FIFO. Два входных FIFO собирают данные с двух входов, третье FIFO через мультиплексор поочередно принимает данные от входных буферов линия за линией, таким образом чтобы выходной поток в первой линии содержал сначала данные от первого источника затем от второго источника и т.д.

Video Output IP. Генерирует поток видеоданных и сигналы синхронизации для Arduino. Записывает данные в dual clock FIFO на системной частоте 100 МГц и выдает наружу на частоте 54МГц. Так же поток дополняется «пустыми» областями – vertical blank и horizontal blank. Размеры этих областей заданы минимальными, т.к. опыты показали, что DCMI приёмник Arduino не чувствителен к динамическому изменению размеру этих областей, таким образом выдача новой строки начинается после того, как пройдет время, указанное в параметре horizontal blank и в FIFO накопится достаточно данных для выдачи, что приводит к небольшим изменением размера пустых областей на кадре.

Тестирование камеры

Ниже представлено первое изображение, полученное с камеры в OpenMV IDE и исходный код phyton скрипта:

Еще один вариант картинки, где на сенсоры установлены разные объективы, на правом изображении фокусное расстояние объектива 2,8мм, а на левом – супертелефото объектив 25мм. На изображении нанесен оверлей для удобства поиска увеличенного изображения на обзорной картинке. Попробуем теперь воспользоваться широкими возможностями OpenMV и определить расстояние до некоего объекта. Простейший способ – так называемый blob detection, когда на равномерном фоне выделяется контрастный объект, и находятся его координаты. Подразумевается, что объект перед камерой один, таким образом мы получим два blob'а находящихся приблизительно на одной вертикальной

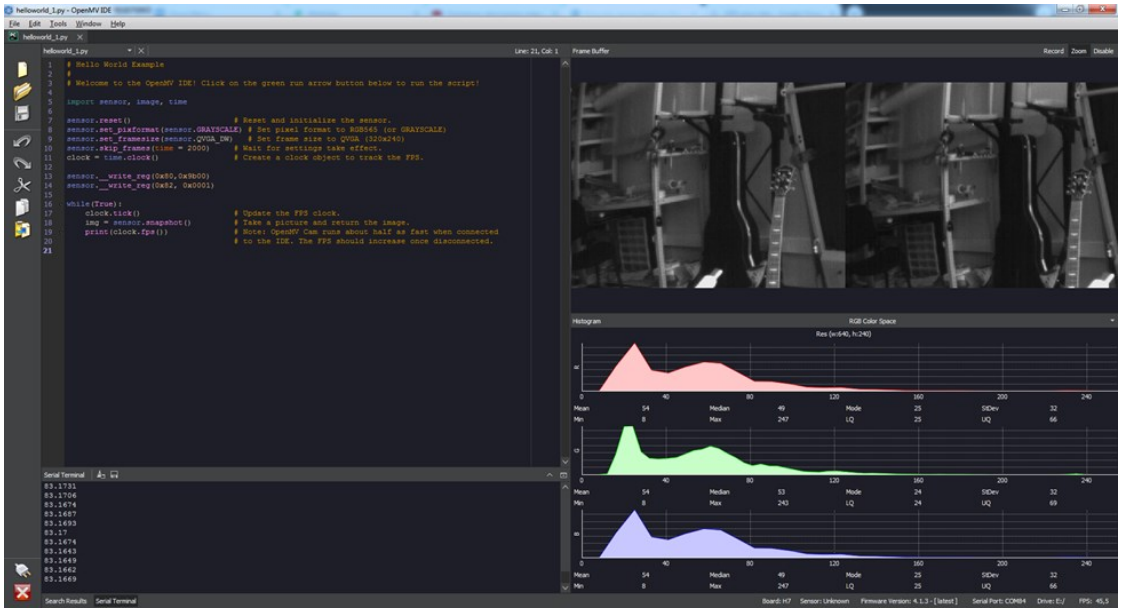


Рисунок 10. Главное окно OpenMV IDE с первым изображением от камеры

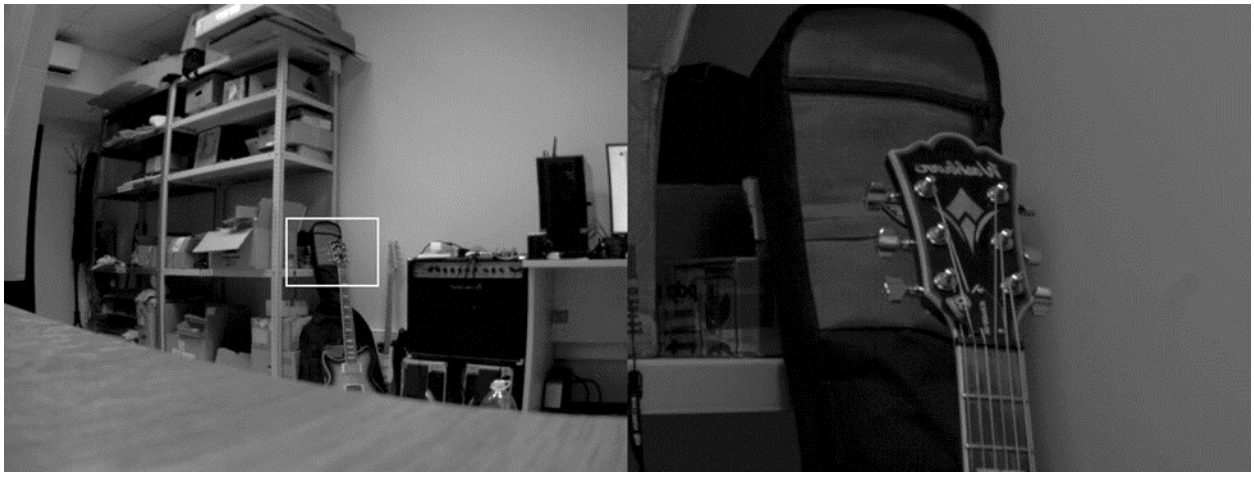


Рисунок 11. Комбинированное обзорное и увеличенное изображение

координате и имеющих смещение по горизонтальной оси, соответствующее расстоянию от камеры до объекта. Истинное расстояние зависит от оптических параметров камеры, но для упрощения мы можем использовать условное «псевдорасстояние», чтобы определять, что камера приблизилась к объекту или удалилась от него:

Рисунок 12. Фактическое расстояние до стакана около 100см

Рисунок 13. Фактическое расстояние до стакана около 60см

Рисунок 14. . Фактическое расстояние до стакана около 30см

Такой метод не требует больших вычислительных ресурсов и на разрешении 1280x480 алгоритм обрабатывал около 14 кадров в секунду (в зависимости от времени экспозиции)

Протестируем возможности работы с нейросетями. Для детектирования и классификации объектов предлагается использовать фреймворк TensorFlow-Lite. Фреймворк позволяет очень просто произвести обучение нейросети по набору изображений и сгенерировать файл для развёртывания сети на микроконтроллере. Я использовал открытый проект с уже размеченным датасетом с изображениями человека, и произвел переобучение с параметрами для Arduino Portenta H7 и монохромного сенсора. Идея заключалась в комбинировании метода обнаружения человека с помощью нейросети на одной половине изображения и поиска соответствующего образа с помощью корреляции на второй половине. Такая комбинация позволит определить на снимке несколько персон, и точно сопоставить их на обоих изображениях. Поскольку используется сенсоры с глобальным затвором, а момент захвата кадра синхронизирован до наносекунды, различий в положении человека на левом и правом кадре не будет даже при быстром движении человека или камеры, а значит метод корреляции должен дать удовлетворительный результат.

Размер удвоенного кадра в этом тесте 960x240. Поиск объекта осуществляется на левом полукадре, причем в его середине, т.к. поиск может осуществляться лишь в квадратной области. Таком образом я ищу изображение человека в середине левом полукадра в области 240 на 240 пикселей. Ниже представлен фрагмент кода скрипта, отвечающий за поиск объектов, соответствующих распознаваемым классам. Класс 0 соответствует фону, при его обнаружении просто пропускаем этот объект.

```
for i, detection_list in enumerate(
        net.detect(img,
roi=NET_ROI,thresholds=[(math.ceil
(min_confidence * 255), 255)])
    ):
        if i == 0:
            continue  # background class
        if len(detection_list) == 0:
            continue  # no detections for
this class?
```

Если изображение человека найдено, я копирую эту часть изображение с небольшим запасом по краям и произвожу поиск на всем втором полукадре методом корреляции:

```
img_template = img.copy(1.0, 1.0, ([x-10, y-
10, w+20, h+20]), copy=True)
r = img.find_template(img_template, 0.7,
step=4 , roi=[WINDOW_FRAME_WIDTH, 0, WIN-
DOW_FRAME_WIDTH, WINDOW_FRAME_HEIGH])
```

Тесты производись на напечатанных на 3D принтере фигурках людей, общий вид эксперимента представлен на фотографии:

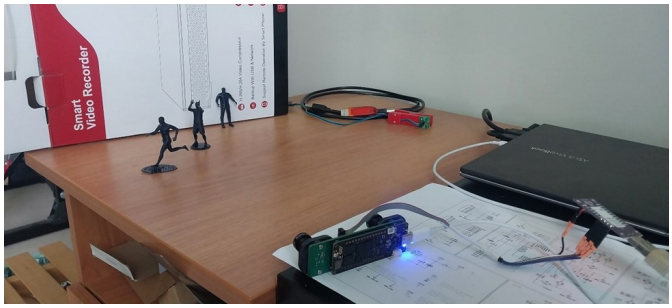


Рисунок 15. Общий вид эксперимента с поиском объектов и расчетом расстояния до них

Три различных фигуры расположены на разном расстоянии от камеры. В результате удалось получить одновременно расстояние до трех различных фигур. Стабильность определения человека средняя, не во всех ракурсах камера стабильно определяла все три фигуры, но точность алгоритма корреляции оказалась удовлетворительной, если фигура была найдена на леовй части кадра, она всегда находилась и на правой

части. Дистанция, как и в эксперименте с blob detection, отображается условно, без пересчета в реальное расстояния.

В данном эксперименте при нахождении трех объектов на правом кадре и, соответственно, трех расчетов корреляции, скорость обработки падала до 4 кадров в секунду.

Планы по доработке устройства.

При тестировании вне помещения была выявлена острая необходимость реализации алгоритма автоматической экспозиции и автоматического контроля усиления. В некоторых случаях полезно было бы иметь модуль гамма-коррекции внутри FPGA прошивки. Все эти функции могут быть успешно реализованы на Arduino, однако это повлияет на производительность основного алгоритма пользователя, к тому же потребуется в каждом новом проекте Arduino добавлять этот функционал, что неудобно для быстрого прототипирования. Так же заложенный чип FPGA обладает запасом ресурсов для реализации предварительной обработки изображения, такого, как например фильтрация или морфологические преобразования. GW2AR имеет встроенную SDRAM память, это позволяет буферизировать кадры, а значит есть возможность реализовать алгоритмы расчета попиксельной разности кадров для определения движения(аналог событийной камеры). Более сложные алгоритмы для реализации в FPGA, такие как коррекция изображения по матрицам калибровки или построение карты глубины требует более серьезного погружения в тему.

Выводы

Разработанная камера уже успешно справляется с некоторыми задачами технического зрения, однако она может быть существенно доработана для конкретных задач путем реализации части алгоритмов обработки исходного изображения внутри FPGA

Литература

- https://boredomprojects.net/index.php/projects/robot-navigation-using-stereo-vision-part-2#h4-2-1-camera
- https://static6.arrow.com/aropdfconversion/efb389d1f9c390cf04346a67dbcda75f31d01f43/deca_user_manual.pdf
- Arduino PRO Portenta H7 Carrier Board (https://github.com/Rufus31415/arduino-pro-portenta-h7-carrier-board?ysclid=m164m2e7p2348024267)
- The Open-Source Machine Vision Project (https://github.com/openmv/openmv/)
- Video and Image Processing Suite User Guide (https://www.intel.com/content/dam/support/us/en/programmable/support-resources/bulk-container/pdfs/literature/ug/archives/ug-vip-18-1.pdf)
- MT9V034 Datasheet https://www.onsemi.com/pdf/datasheet/mt9v034-d.pdf
- MT9V024 Slave Exposure Mode Operation https://files.niemo.de/aptina_pdfs/TN-09-283_MT9V024_Slave_Exposure_Mode_Operation.pdf

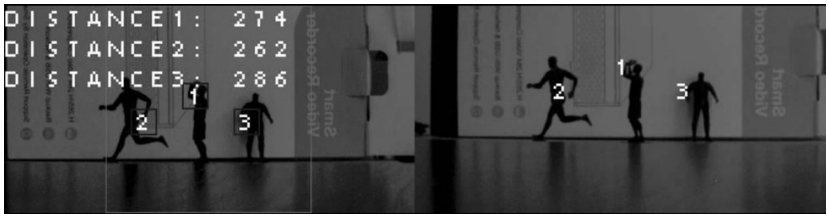


Рисунок 16. Поиск объектов и расчет расстояния. Изображение из OpenMV IDE

Криптопроцессор на FPGA

Александр Хлуденьков и команда «Криптозавры»

Обсуждение и комментарии: link

Введение

В октябре этого года проходил хакатон по разработке криптографических устройств Security Gadget Challenge (https://securitygc.ru). Команда «Криптозавры» под предводительством автора данной статьи выступила с концептом криптопроцессора на FPGA. Мы выступали в финале!

Реализация криптопроцессора на FPGA или ASIC представляет определенный интерес, так как на микросхемах данного типа возможно реализовать быстрое сложение и умножение больших чисел (буквально до тысяч бит), а также провести конвейеризацию данных операций. Так как современные операции шифрования и дешифрования (к примеру, шифр RSA) – это операции перемножения больших бинарных чисел, то становится актуальной задача распараллеливания арифметических процессов на специальном устройстве.

Тема криптографии на FPGA широко обсуждается на просторах интернета и страницах научных журналов. Но, в основном, описываются реализации криптографических ядер для ускорения процессов вычислений. Реализации законченного комплексного устройства обнаружено не было. «Криптозавры» выступили с предложением изготовить криптографическое устройство вместе со всеми портами в одном небольшом корпусе. Концептуально это будет SoC (System on Chip, система на кристалле).

Внутреннюю структуру ASIC-микросхемы практически невозможно прочитать, только построив таблицу истинности для входов и выходов, что для конечного автомата с большим числом состояний практически нереально. Внедрение вредоносного эксплойта после проектирования так же невозможно.

Криптография

Вначале кратко пробежимся по курсу криптографии. По большому счёту, все шифры делятся на симметричные (с закрытым ключом) и асимметричные (с открытым ключом).

Симметричные шифры. Для шифрования и дешифрования используется один и тот же ключ (процессы симметричны, зеркальны). Так как ключ один как для шифрования, так и для дешифрования, то его можно передавать только тем лицам, кто производит шифрование данного сообщения и кому предназначено зашифрованное сообщение. Поэтому они и называются «системы с закрытым ключом». Типичными примерами данных систем являются шифр подстановки и шифр перестановки.

Ассиметричные шифры. Для шифрования и дешифрования используются различные ключи – публичный (открытый) и приватный (закрытый). Сама система основана на различии в сложности выполнения некоторой прямой и обратной математической функции. В данном случае это операции перемножения двух простых чисел (прямая операция) и факторизации, разложения (обратная операция) получившегося произведения. Перемножить два 40-ка разрядных десятичных числа возможно, найти два неизвестных простых сомножителя 80-ти разрядного десятичного числа пока на грани фантастики. Типичными системами шифров с открытым ключом являются системы RSA и эллиптических кривых. Открытый ключ для шифрования можно передавать всем, закрытый ключ – только у того, кто расшифровывает зашифрованное сообщение.

Структура криптопроцессора

После изучения соответствующей математической части было решено проработать реализацию криптопроцессора для шифрования четырьмя различными видами шифра. В начале процесса шифрования необходимо ввести в криптопроцессор вид выбранного шифра и ключ. Так же после процесса шифрования сообщения необходимо зашифрованное сообщение вывести «наружу». Соответственно, необходимо реализовать системы ввода и вывода. Получается общая структура криптопроцессора - модуль ввода, криптоядро и модуль вывода (рисунок 1).

Модули ввода и вывода

Рассмотрим вначале модули ввода и вывода. Структура модуля ввода отображена на рисунке 2.

Входной порт — это всем известные интерфейсы SPI, Ethernet, UART и USB. Порт вывода так же поддерживает перечисленные интерфейсы. Для согласования работы системы ввода-вывода и криптоядра по времени необходимо реализовать 2 очереди

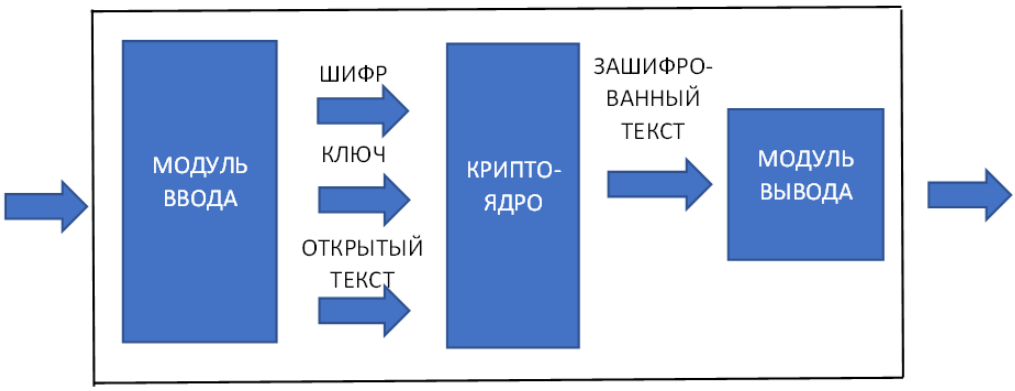


Рисунок 1. Структура криптопроцессора.

FIFO. В общем, возможная периферия упирается в мощность микросхемы и выбранную плату (Таблица 1).

Криптоядро

Так как возможен выбор параметров шифрования, криптоядро содержит несколько подмодулей, каждый из которых реализует свой алгоритм шифрования. Рассматриваются следующие виды шифров: подстановки, перестановки, RSA, эллиптических кривых. Структура криптоядра представлена на Рисунке 3.

Шифр подстановки (замены)

Данный шифр самый простой как для самой криптографии, так и для её реализации на FPGA. При шифровании в сообщении вместо одних символов происходит подстановка других в соответствии с заранее заданной таблицей соответствия. Наиболее известными из них являются шифр Цезаря и шифр Хилла.

Ключом является список («Символ_ОА» → «Символ_ЗА»), где «Символ_ОА» → символ из алфавита открытого текста, «Символ_ЗА» - символ из алфавита закрытого текста. К примеру, символы шифруемого текста заменяются в соответствии с таблицей 2.

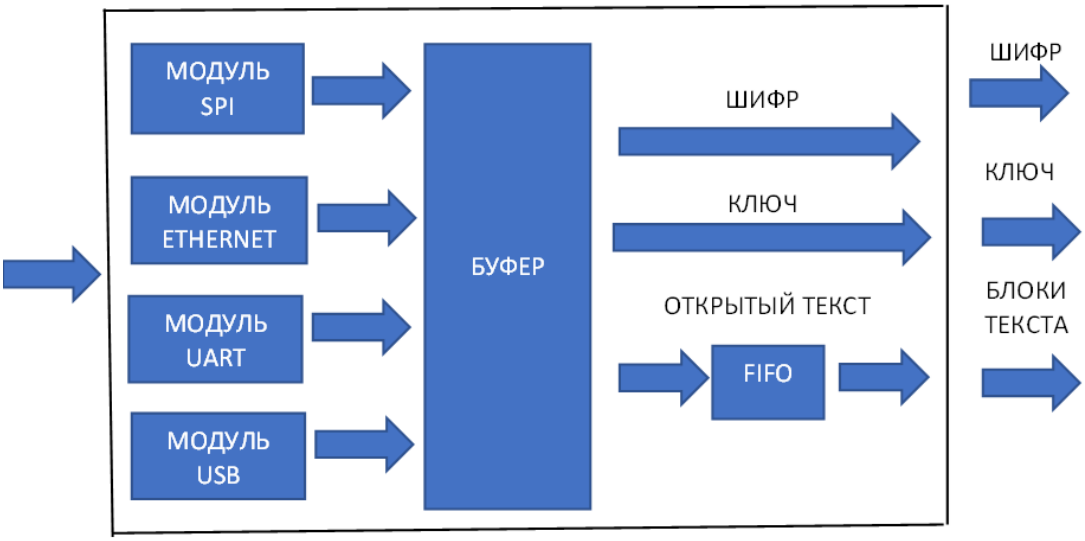


Рисунок 2. Структура модуля ввода.

Таблица 1. Список интерфейсов и рекомендуемых микросхем

Интерфейс	Скорость	Размер программы	Микросхема
I2C	100 кб/сек	1 кБ	Cyclone III EP3C10E144C8
SPI	50 Мб/сек	2 кБ	Max II EPM240T100C5N
UART	921 кб/сек	2 кБ	Max II EPM240T100C5N
Ethernet	100 Мбит/с	3 кБ	Stratix 10

Таблица 2. Ключ шифра подстановки

А	Б	В	Г	Д	Е	Ж	З	И	К	Л	М	Н	О	П	Р	С	Т	У	Ф	Х	Ц
11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
П	К	Е	Н	Г	Ш	Ф	Ы	В	А	Д	Р	Л	Ж	Э	Я	Ч	С	М	И	Т	Ю
55	43	63	78	92	36	71	82	93	12	74	54	33	74	15	29	68	78	38	58	19	26

Тогда слово «ПРОЦЕССОР» (код – 25 26 24 32 16 27 27 24 26) можно будет зашифровать как «ЭЯЖЮШЧЧЖЯ» (код – 15 29 74 26 36 29 29 74 29). Понятно, почему нет чисел, начинающихся с «0», так как, к примеру, число «07» – может просто замениться на «7». Процесс замены символов будет выглядеть так:

```
module Substitution (input [7:0] planeText,
output [7:0] shifrText);
always @(planeText)
case (planeText)
        11: shifrText = 55;
        12: shifrText = 43;
        13: shifrText = 63;
         . . .
         . . .

default: shifrText= 11;
endcase
```

Процесс дешифрования происходит в обратном порядке.

Шифр перестановки.

Шифр перестановки - метод перемены мест символов в открытом тексте. Является более стойким к криптоанализу, чем подстановки. Ключом является определённый способ перестановки знаков в пределах одного блока.

Ключом является список «Позиция_ОТ» → «Позиция_ЗТ», где «Позиция_ОТ» - позиция символа открытого текста в пределах блока, «Позиция_ЗТ» - позиция символа закрытого текста в пределах блока. К примеру, для блока длиной 8 символов (64 бита) ключом может быть такой:

(1 → 5, 2 → 3, 3 → 8, 4 →7, 5 →2, 6 → 1, 7 → 5, 8 → 4).

Индексацию элементов лучше вести с «нуля», т. е. первый элемент это a_0, второй a_1 и т. д. Тогда ключ перестановок (0 → 2, 1 → 3, 2 → 1, 1 → 0) в двоичном виде будет выглядеть: (00 → 10, 01 → 11, 10 → 01, 11 → 00). Под ключ резервируется массив типа wire:

```
wire [1:0] [N:0] key; // где N – размер
блока.
```

Шифр перестановки при реализации на схеме FPGA сложнее, чем шифр подстановки. Не существует способа «на лету» менять контакты в железе. Процесс перестановки можно заменить выборкой логическим умножением на необходимый ключ, выбранный из таблицы.

Таблица 3. Ключ для шифра перестановки

	00	01	10	11
00	0	0	1	0
01	0	0	0	1
10	0	1	0	0
11	1	0	0	0

В данной таблице в ячейках значения «1» означают, что данные значения ячеек присваивается от обозначенных слева в обозначенные сверху, значение «0» - что нет.

Шифр RSA

Предыдущие два шифра являются шифрами с закрытым ключом. Рассмотрим один из самых известных шифров с открытым ключом – RSA.

Данный шифр основан на сложности факторизации (разложения на множители) числа, являющегося произведением двух простых чисел. Для реализации алгоритма RSA используются очень большие простые числа до 256 бит каждое (примерно 80 десятичных знаков). Произведение получается размером около 160 десятичных знаков. При должном выборе множителей (простых чисел) факторизовать данное число, не зная один из сомножителей, практически нереально.

Существуют алгоритмы быстрого умножения, (Карацубы и Шенхаге-Штрассена), их возможно организовать аппаратно на FPGA. Так как нам не важны получающиеся биты переноса для старшего бита, то умножение можно реализовать в виде сдвига со сложением. Для этого используется сдвиговый регистр и сумматор.

Шифр на эллиптических кривых

Так же, как и RSA, шифр на эллиптических кривых является шифром с открытым ключом. Процесс шифрования (дешифрования) основан на целочисленном решении уравнения вида: $y^2 = x^3 + ax + b \pmod p$. Призван уменьшить длину ключа при той же стойкости шифра, что и RSA.

Выбор платы

Процесс выбора платы было решено одновременно прорабатывать в двух направлениях – бюджетном и топовом. Для варианта с бюджетным исполнением была выбрана плата Intel DE10-Lite как имеющаяся у команды (рисунок 4). Данная плата используется для занятий «Школы синтеза цифровых схем».

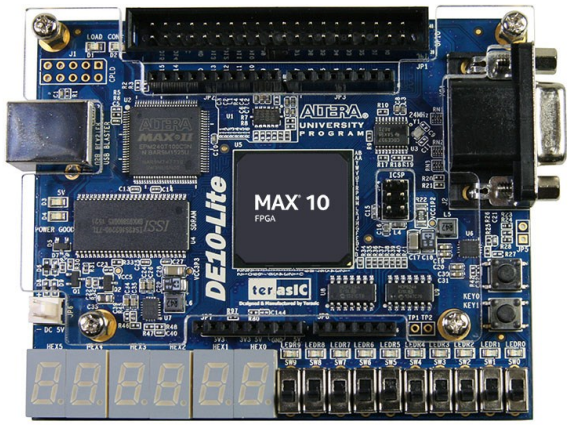


Рисунок 4. Отладочная FPGA-плата DE10-Lite.

Технические параметры данной платы представлены в таблице 4.

Таблица 4. Параметры платы DE10-Lite

№	Параметр	Значение
1	Чип	MAX-10 10M50DAF484C7G
2	Тип	FPGA
3	Количество элементов	50 000
4	Тактовая частота	50 МГц
5	Встроенные умножители	144 – 18x18 бит
6	Количество контактов (чип)	484
6	Количество GPIO (плата)	2x20 (3,5 V)
7	Встроенные порты (плата)	Порт VGA

Для отработки варианта «Топового криптопроцессора» была выбрана плата Stratix-10 (рисунок 5). Это одна из «топовых» микросхем фирмы Altera-Intel. Данная микросхема имеет следующие характеристики:

- Кристалл: Stratix-10;
- Тактовая частота: 1 ГГц;
- Количество элементов: 2,8 млн.

Рисунок 5. FPGA-плата Stratix-10

Тестирование криптопроцессора

Отладка и тестирование криптопроцессора на FPGA производится на

Пользователь вводит на компьютере в GUI параметры шифрования – выбирает шифр и его параметры - размер ключа и сам ключ. После ввода параметров шифра и ключа пользователь вводит текст для шифрования (plaintext). Введенные данные через COM-порт попадают в плату Ардуино. Ардуино переводит текст в бинарную последовательность и передает по интерфейсу SPI в криптопроцессор.

Рассмотренное выше устройство на FPGA-микросхеме MAX-10 является тестовым, так как скорости работы микросхемы MAX-10 недостаточно для хорошей обработки данных. Для «боевой» работы необходимо использовать намного более сложные микросхемы, такие как Stratix-10. В конечном результате необходимо разрабатывать собственную плату на соответствующем чипе.

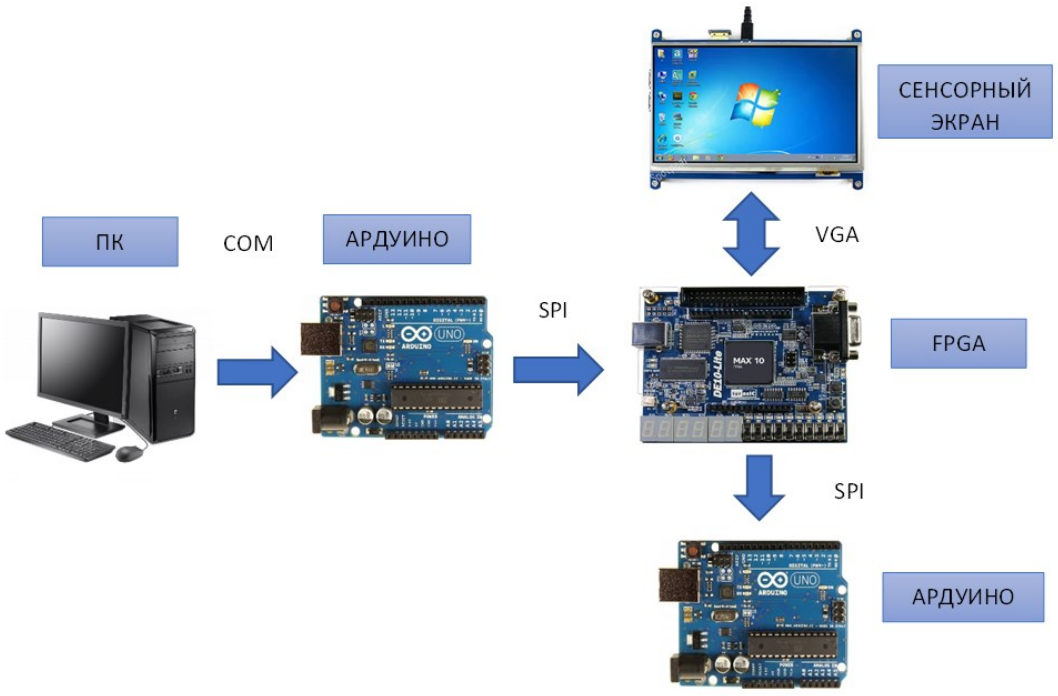


Рисунок 6. Стенд тестирования криптопроцессора

Особенности разработки аппаратного LDPC кодера

Лотник Виталий

Обсуждение и комментарии: link

База

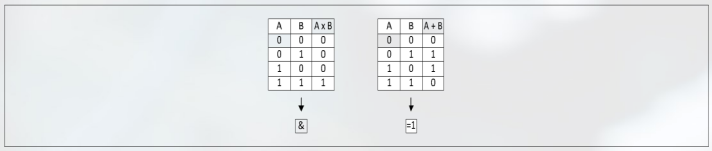
Помехоустойчивое кодирование - добавление к передаваемой информации дополнительных данных с целью обнаружения и исправления ошибок при приеме.

Скорость кода - отношение длины исходной информационной последовательности к длине кодированной последовательности. Например, при скорости кода 1/2 кодированная последовательность в два раза длиннее исходной.

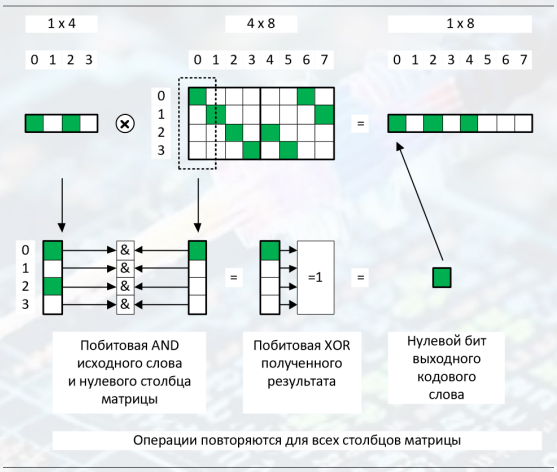
LDPC (Low-Density Parity-Check) - код с малой плотностью проверок на чётность. Процедура LDPC кодирования заключается в перемножении информационной последовательности I на некоторую матрицу G:

$$C = I \times G$$

В аппаратной реализации матричное умножение вырождается в битовые операции AND и XOR.



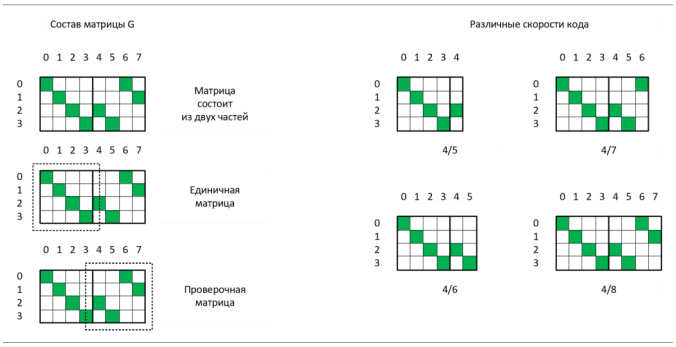
Процесс формирования выходного кодового слова выглядит следующим образом:



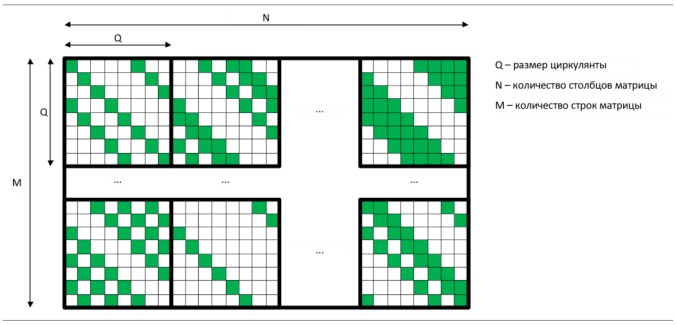
Матрица G состоит из двух частей: единичная и проверочная.

В единичной матрице все биты, расположенные на главной диагонали, равны единице, остальные - нулю. Это приводит к тому, что первая часть выходного кодового слова совпадает с исходной информационной последовательностью.

Количество столбцов в проверочной матрице определяет скорость кода. Количество строк - совпадает с длиной исходной информационной последовательности.



Проверочные матрицы, используемые при реализации разрабатываемого кодера, имеют особенность: они состоят из блоков - циркулянт. Строки каждой циркулянты формируются путем циклического сдвига предыдущей строки на один бит.



Исходными данными для разработки аппаратного кодера являются:

- размеры входных информационных слов (до нескольких тысяч бит);
- скорости кода;
- файлы с проверочными матрицами.

Получение матриц

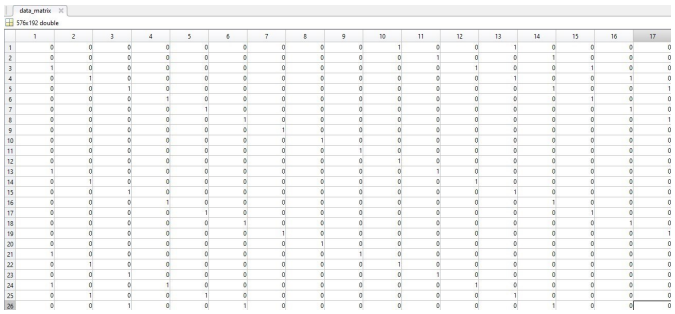
В моем случае матрицы записаны в файлы в битовом виде. Для чтения данных из файла и дальнейшей обработки используется Matlab:

```
file_name_rx = 'LDPC_matrix.mat';
fid_rx = fopen(file_name_rx);
data_rx = fread(fid_rx, 'ubit1');
```

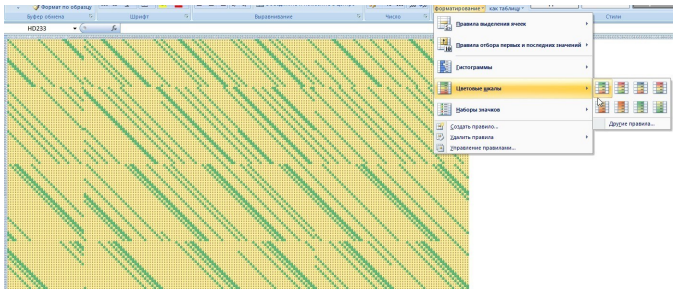
В результате будет сформирована переменная data_rx размером (N*M x 1). Для формирования матрицы используется команда reshape:

```
data_matrix = reshape(data_rx, M, N);
```

В результате будет сформирована переменная data_matrix размером (M x N).



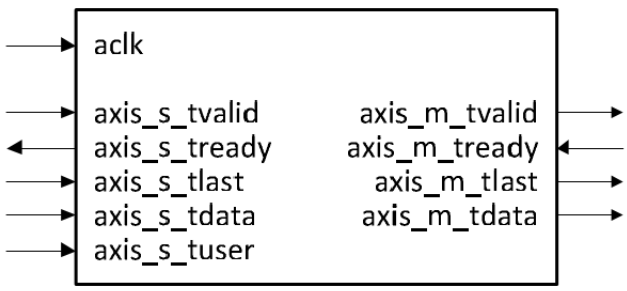
Для визуализации матрицы можно скопировать ее в Excel:



Полученный массив в дальнейшем будет использоваться для кодогенерации.

Интерфейс кодера

Для реализации кодера используется AXI Stream интерфейс:



Перечень портов:

название	напр.	разр.	описание
aclk	input	1	тактовый сигнал
axis_s_tvalid	input	1	валидность входных данных
axis_s_tready	output	1	готовность принять следующий отсчет входных данных
axis_s_tlast	input	1	признак последнего отсчета входных
axis_s_tdata	input	8	входные данные
axis_s_tuser	input	8	конфигурация кодера
axis_m_tvalid	output	1	валидность выходных данных
axis_m_tready	input	1	готовность выдать следующий отсчет выходных данных
axis_m_tlast	output	1	признак последнего отсчета выходных
axis_m_tdata	output	8	выходные данные

На вход кодера через порт axis_s_tdata поступают байты информационного слова, которое должно быть закодировано (размеры кодируемых слов должны быть кратны байту). Каждый байт сопровождается признаком axis_s_tvalid, при этом следующий байт может подаваться только при активном axis_s_tready. Последний байт информационного слова сопровождается признаком axis_s_tlast. Через порт axis_s_tuser задается конфигурация кодера (скорость кода, размер циркулянты) для текущего информационного слова.

Байты закодированного слова выдаются на порт axis_m_tdata. Каждый байт сопровождается признаком axis_m_tvalid, при этом следующий байт может выдаваться только при активном axis_m_tready. Последний байт закодированного слова сопровождается признаком axis_m_tlast.

Порядок бит во входном байте таков, что биты информационного слова занимают позиции, начиная с нулевой:

Обратное преобразование выполняется для выходного байта и закодированного слова:

Логика работы

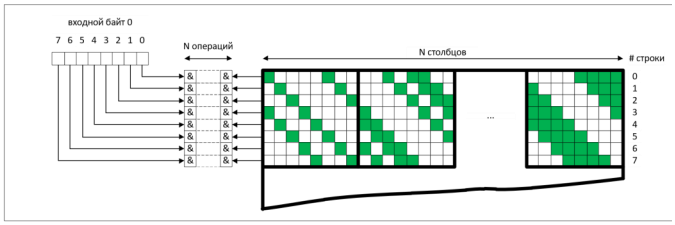
Работа кодера состоит из двух этапов:

Трансляция входных данных на выход для формирования первой части кодера. Параллельно с этим процессом производится умножение входных данных на проверочную матрицу.

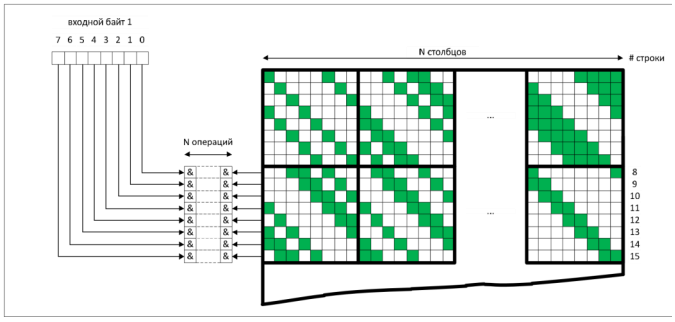
Выдача данных, полученных в результате умножения.

Данные на вход приходят побайтово, соответственно, для каждого нового отсчета необходимо выполнить операции AND с восемью строками проверочной матрицы. Суммарное количество операций AND будет равно N*8, где N - количество столбцов проверочной матрицы.

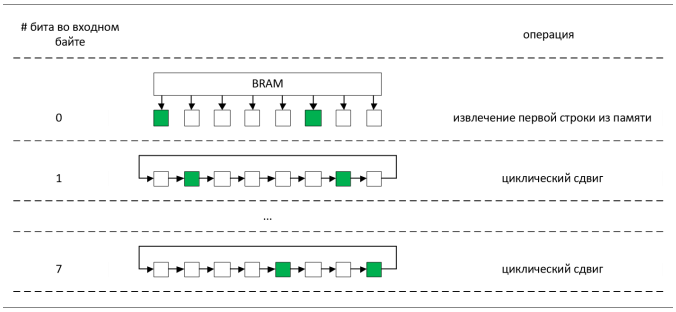
Обработка байта #0:



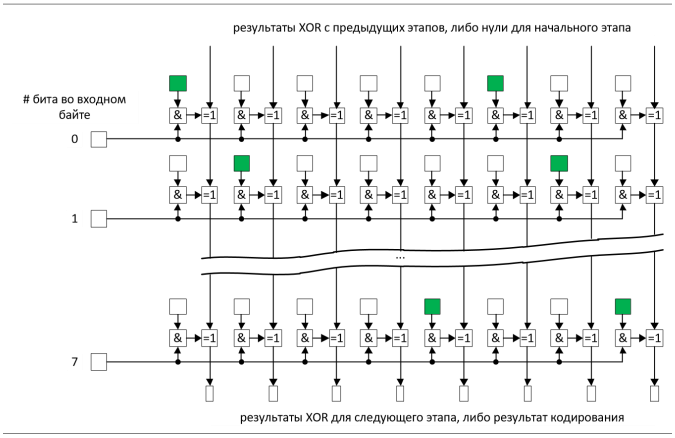
Обработка байта #1:



Используемые матрицы хранятся во внутренней памяти ПЛИС. С учетом, что матрицы состоят из циркулянт, память можно сэкономить, если хранить только их первые строки, а следующие строки получать путем циклического сдвига. Тогда алгоритм формирования одной циркулянты для одного байта будет следующим:



Кроме того, необходимо построчно выполнить операции XOR полученных результатов. Итоговая схема обработки одного байта данных одной циркулянтой будет иметь вид:



Пример кода для выполнения данных операций:

```
always @(posedge ib_clk) begin
    // цикл по количеству бит в байте
    for (int b_id = 0; b_id < 8; b_id++) begin
        // цикл по количеству циркулянт
        for (int circ_id = 0; circ_id < CIRC_NUM; circ_id++) begin
            if (b_id == 0 && circ_row_id == 0) begin
                // для нулевого бита и первой строки циркулянты извлекаем строку матрицы из памяти
                coder_sreg[circ_id] = coder_ldpc_table[block_id*CIRC_NUM + circ_id];
            end else begin
                // для остальных битов выполняем циклический сдвиг
                coder_sreg[circ_id] = {coder_sreg[circ_id][7], coder_sreg[circ_id][0:6]};
            end
            // цикл по количеству бит в байте
            for (int jj = 0; jj < 8; jj++) begin
                // выполняем операции AND
                coder_and_reg[circ_id][jj] = coder_sreg[circ_id][jj] && input_tdata[b_id];
                // для нулевого бита нулевого входного байта выполняем AND
                if (b_id == 0 && block_counter == 0) begin
                    coder_xor_reg[circ_id][jj] = coder_and_reg[circ_id][jj];
                // для остальных битов выполняем AND и XOR с результатом с предыдущего этапа
                end else begin
                    coder_xor_reg[circ_id][jj] = coder_and_reg[circ_id][jj] ^ coder_xor_reg[circ_id][jj];
                end
            end
        end
    end
```

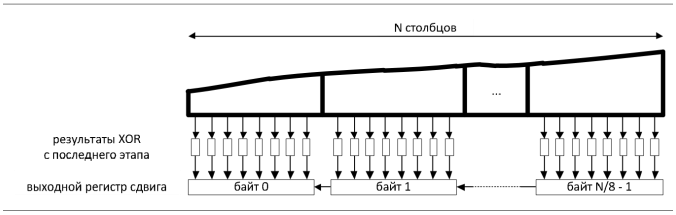
```
    // цикл по количеству циркулянт, итоговый результат
    for (int circ_id = 0; circ_id < CIRC_NUM; circ_id++) begin
        coder_result[circ_id] <= coder_xor_reg[circ_id];
    end
end
```

Здесь входной байт обрабатывается за один такт. Для этого реализован цикл на восемь итераций, в котором выполняется извлечение строки матрицы из памяти, ее последующий циклический сдвиг и необходимые операции AND и XOR для всех битов.

В целом такой код синтезируется, однако, при увеличении размеров матрицы и повышении сложности кодера, занимает большое количество ресурсов.

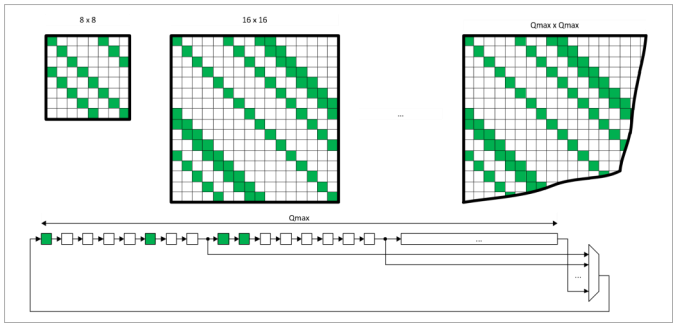
Для уменьшения количества используемых ресурсов можно перейти к схеме, в которой входной байт обрабатывается за восемь тактов, пожертвовав пропускной способностью кодера.

Результат обработки всех байтов входного информационного слова поступает в регистр сдвига длиной N бит, после чего побайтово выдается на выход.

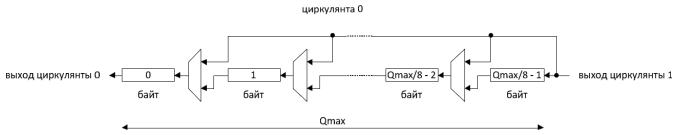


Особенности реализации при различных параметрах кода.

Матрицы могут содержать циркулянты различных размеров, например 8x8, 16x16, 24x24 и т.д. Это приводит к тому, что необходимо выполнять циклический сдвиг разного размера. Схемотехнически это вырождается в регистр сдвига, длина которого равна максимальной длине строки циркулянты. С разных отводов данного регистра сигналы поступают на выход мультиплексора, а с выхода мультиплексора - на начало регистра.



Размер циркулянты также влияет на выходной регистр сдвига. Он разбивается на байты и в рамках одной циркулянты на вход каждого байта поступают либо данные со следующего байта этой же циркулянты; либо данные с выхода следующей циркулянты (за исключением последнего байта - на него всегда приходят данные с выхода следующей циркулянты).



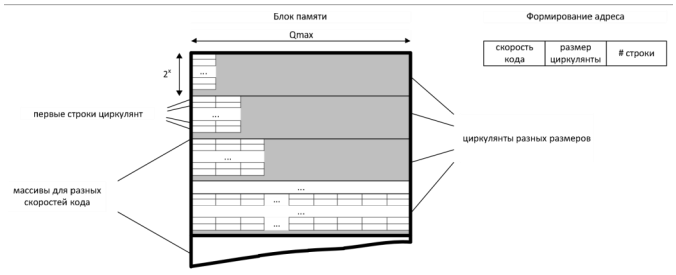
Особенности организации памяти для хранения матриц

Хранение матриц в памяти осуществляется исходя из следующих условий:

необходимо хранить первые строки циркулянт;

циркулянты могут быть разного размера;

для разных скоростей кода используются разные матрицы.



Блок памяти заполняется первыми

строками циркулянт наименьшего размера (1 байт). Для этого выделяются 2^x адресов (если количество строк не кратно степени двойки, некоторые адреса не используются). Следующие 2^x адресов заполняются первыми строками циркулянт размером 2 байта, затем - 3 байта и так далее вплоть до максимального размера Qmax. Аналогично заполняются строки памяти для остальных скоростей кода.

Для выбора текущей строки из памяти формируется адрес, состоящий из трех частей: скорость кода (старшие биты), размер циркулянты и номер строки (младшие биты).

Исследование эффективности верификации с использованием PyUVM и SystemVerilog-UVM

Свинцов А.А.
магистр НИУ МИЭТ
andrej.svin@yandex.ru

Обсуждение и комментарии: link

Аннотация

Python, как мультипарадигмальный язык, известный своей простотой интеграции с другими языками, в последнее время завоевал значительное внимание среди инженеров по верификации. Среда верификации на базе Python использует такие открытые библиотеки, как PyUVM, обеспечивающая реализацию UVM 1.2 на базе Python, и PyVSC, способствующая рандомизации с ограничениями и функциональному покрытию. Эти библиотеки играют ключевую роль в ускорении разработки тестов и перспективны для снижения затрат на настройку. Целью данной работы является оценка эффективности верификации цифровых дизайнов с помощью PyUVM и сравнение возможностей и показателей производительности с устоявшейся методологией SystemVerilog-UVM.

Ключевые слова: верификация, uvm, pyuvm, python, systemverilog, цифровые схемы, оптимизация.

Введение

По мере непрерывного роста сложности проектов систем на кристалле верификация становится всё более сложной задачей. В результате время, необходимое для верификации, значительно увеличивается. Кроме того, возникает необходимость в более продуктивной и эффективной работе при ограниченном количестве сотрудников. Методологии, применяемые в отрасли, такие как ограниченная случайная верификация, формальная верификация и верификация на основе метрик, используют SystemVerilog в качестве языка описания аппаратуры. Этот язык предлагает множество возможностей, включая объектно-ориентированное программирование и функциональное покрытие, однако его изучение требует значительных усилий, особенно для начинающих.

Различия Python и SystemVerilog

SystemVerilog имеет высокий порог вхождения по сравнению с другими языками программирования. В связи с этим UVM,

основанный на SystemVerilog, становится более сложным в использовании. С другой стороны, Python обладает низким порогом вхождения, является одним из самых популярных языков и легко интегрируется с такими библиотеками, как Numpy, Pandas и др. Согласно исследованию исследовательской группы Уилсона, 23% всех проектов специализированных интегральных схем используют Python для различных задач.

Замечание по таблице для Python:

- Позволяет не объявлять переменные и выполнять операции над ними;
- Поддерживает сложные структуры данных, например, кортежи и словари;
- Исключения обрабатываются с использованием блоков try, except и finally;
- Легче создать эталонные модели для дизайна, благодаря поддержке множества библиотек;
- Позволяет перезапустить симулятор без перекомпиляции и редактировать тесты во время его работы.

Замечание по таблице для SystemVerilog:

- Если размер не объявлен то, данные могут быть потеряны после присваивания к переменной другого размера;
- Функции не являются объектами и не могут быть сохранены или переданы напрямую в качестве аргументов.

Время выполнения тестовых сценариев PyUVM медленнее, чем у тестовых сценариев SystemVerilog-UVM. Такое различие во времени выполнения тестовых сценариев связано с их разными подходами. В SystemVerilog для установления связи с симулятором используются директивы и команды моделирования, что приводит к тесной интеграции, которая улучшает выполнение и сокращает время выполнения. В отличие от них, Python взаимодействуют с

Таблица 1 – Сравнение SystemVerilog и Python в верификации

Характеристика	SystemVerilog	Python
Декларация типов данных	Статическая	Динамическая
Поддерживаемые типы логики	0, 1, X, Z	X, Z, U, W
Параметризация и размер переменной	Требуется	Не требуется
Стиль контроля потока	begin, end	Правильный отступ
Функции	Не объекты	Вызываемые объекты
Исключения	Не поддерживаются	Поддерживаются
Библиотеки	-	Поддерживаются
Интерпретируемый	Нет	Да
Иерархия дизайна	включает верхний testbench	Не включает верхний testbench

симулятором с помощью VPI/VHPI, которые менее тесно интегрированы. Эти издержки становятся более значительными по мере увеличения количества транзакций, что приводит к увеличению времени выполнения симуляции (см. рисунок 1).

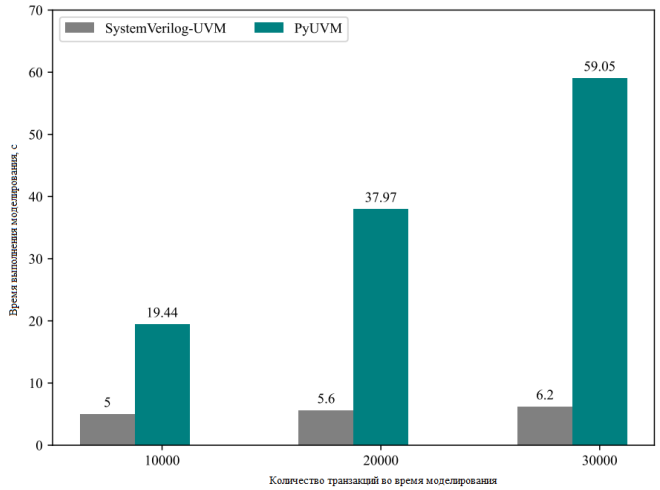


Рисунок 1 – Время тестирования с разным количеством транзакций

С точки зрения использования оперативной памяти, Python потребляет на 30 -35% больше памяти. В симуляторе Questa Sim при длительных симуляциях наблюдаются утечки памяти в Python, где потребление памяти могло быть в несколько раз выше, чем в SystemVerilog (см. рисунок 2).

Покрытие в Python оказалось меньше на 1,6%, разница начала проявляется после 400 операций. Это показывает, что по мере увеличения количества операций, PyVSC чаще выдавал случайные значения, которые уже были обработаны (см. рисунок 3).

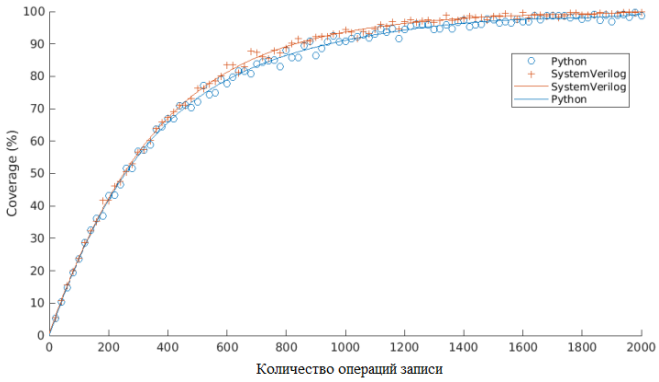


Рисунок 3 – Покрытие от операций записи для Python и SystemVerilog

Можно выделить то, что один и тот же тест в SystemVerilog имеет больше строк кода. Важно отметить, что увеличение количества

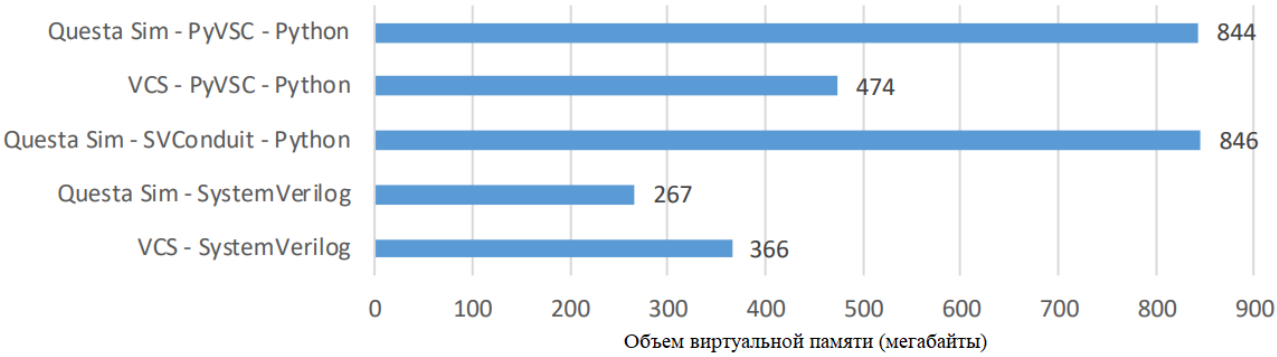


Рисунок 2 – Потребление памяти в разных симуляторах

SV:

```
class ymp_lpi_q_agent extends uvm_agent;
  ymp_lpi_q_agent_cfg cfg;
  ymp_lpi_q_monitor   monitor;
  ymp_lpi_q_driver    driver;
  ymp_lpi_q_sequencer sequencer;

  extern function new(string name, uvm_component parent);
  extern function void build_phase(uvm_phase phase);
  extern function void connect_phase(uvm_phase phase);

  `uvm_component_utils(ymp_lpi_q_agent)
endclass

function ymp_lpi_q_agent::new(string name, uvm_component parent);
  super.new(name, parent);
endfunction

...

endclass
```

Python:

```
class lpi_master_agent(uvm_agent):
  num_id = 0

def __init__(self, name, parent, *args, **kwargs):
  lpi_master_agent.num_id +=1
  self.num_id = lpi_master_agent.num_id
  super().__init__(name, parent)
```

Рисунок 4 – Примеры объема кода на SystemVerilog и Python

строк кода может влиять на читаемость, поддержку и тестирование программных решений (см. рисунок 4).

Заключение

Исследование верификации DUT с использованием окружений написанных на PyUVM и SystemVerilog-UVM проводилось по различным параметрам (объем кода, время симуляции, потребления памяти, покрытие). Несмотря на то, что моделирование на Python занимает больше времени и потребляет больше памяти, оно может быть более эффективным при условии, что генерация тактовых импульсов перенесена на сторону DUT. Моделирование с помощью PyUVM позволяет собирать входные данные и покрытии в удобном формате. Их можно проанализировать для создания новых методологий на основе методов машинного обучения, которые ещё больше ускорят процесс верификации.

Список литературы

1. H. Foster, “2022 Wilson Research Group Functional Verification Study,” Siemens Digital Industries Software, Tech. Rep., Oct. 2022.
2. D. Gadde, S. Kumari, A. Kumar, “Towards Efficient Design Verification -- Constrained Random Verification using PyUVM”, Cornell University, May 2024.
3. M. Sinerva, “UVM testbench in Python: feature and performance comparison with SystemVerilog implementation”, University of Oulu, June 2023.
4. Quinn, “Constrained Random Stimulus Generation using Python,” DVClub Europe, 2021.
5. M. DSU, PY-UVM Framework for RISC-V Single Cycle Core, May 8, 2023 (Accessed: August 4, 2023).

БИХ-ФИЛЬТРЫ: ОСНОВНЫЕ ПОНЯТИЯ, ФОРМЫ И РАСЧЕТ

Бабаев Рашид Эльдарович

Telegram: @skand_en

e-mail: rashidbabaev123@gmail.com

Обсуждение и комментарии: link

Аннотация

Статья посвящена реализации фильтра с бесконечной импульсной характеристикой (БИХ, рекурсивный фильтр) на языке описания аппаратуры Verilog. В работе коснемся теоретические основы БИХ-фильтров, включая их математические модели, а также особенности различных форма построения. Проведен анализ амплитудно- и фазочастотных характеристик (АЧХ и ФЧХ) для конкретного типа БИХ-фильтра, их устойчивости, вычислительной эффективности и требований к аппаратным ресурсам. Обсудим практические аспекты построения и оптимизации RTL БИХ-фильтров, требующих высокой производительности и точности фильтрации, на уровне аппаратуры.

Теоретические основы БИХ-фильтров

БИХ-фильтр (фильтр с бесконечной импульсной характеристикой или рекурсивный фильтр) — это один из классов цифровых фильтров, используемых в обработке сигналов, отличительной особенностью которого является его импульсная характеристика. В отличие от КИХ-фильтров (фильтров с конечно импульсной характеристикой или нерекурсивных), БИХ-фильтры обладают обратной связью, которая позволяет им реагировать на входные данные в течение неограниченного времени, формируя бесконечные выходные данные. По этой причине БИХ-фильтры могут быть не устойчивыми. Однако, это особенность делает их особенно полезными при построении фильтров с резкими перепадами АЧХ задействуя меньше вычислительной мощности и аппаратных ресурсов.

Математически БИХ-фильтр описывается разностным уравнением, в котором выходной сигнал в определяется не только текущим и прошлыми значениями линии задержки, но и прошлыми значениями этого выходного сигнала. Это уравнение записывается в виде:

$$y[k] = \sum_{m=0}^{M} b_k x[k-m] - \sum_{n=1}^{N} a_k y[k-n]$$

где y[k - n] — выходной сигнал с n-ой задержкой, x[k - m] — входной сигнал с m-ой задержкой, b_k и a_k — коэффициенты фильтра, а M и N — их максимальные порядки.

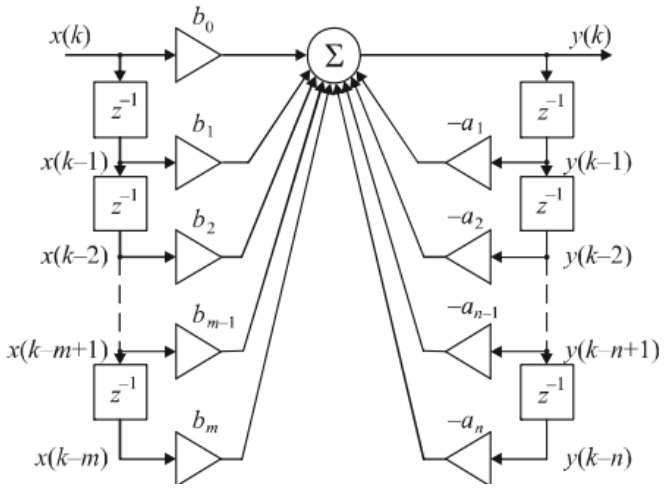


Рисунок 1. Прямая реализация БИХ-фильтра

Благодаря наличию обратной связи и меньшему количеству коэффициентов по сравнению с КИХ-фильтрами для достижения аналогичной частотной характеристики, БИХ-фильтры требуют значительно меньших вычислительных ресурсов. Это делает их предпочтительными для задач, где важна высокая вычислительная эффективность и ограничены аппаратные ресурсы.

Полюса и нули передаточной функции играют важную роль в характеристиках БИХ-фильтра. Поскольку полюса могут находиться внутри или на границе единичной окружности комплексной плоскости, малейшее отклонение от устойчивого состояния может привести к значительным искажением сигнала или к нестабильности фильтра. Поэтому при проектировании БИХ-фильтров особое внимание уделяется анализу устойчивости и расположению полюсов, что критически важно для надежности их работы.

При проектировании фильтра отдельно внимание уделяется устойчивости фильтра. Полюса функции передачи должны находиться внутри единичного круга комплексной плоскости. В случае не устойчивости рекурсивного фильтра может привести к искажению выходного сигнала и нестабильной работе.

Формы реализации БИХ-фильтра

Формы реализации дискретных фильтров: прямая, каноническая, транспонированная, параллельная и последовательная.

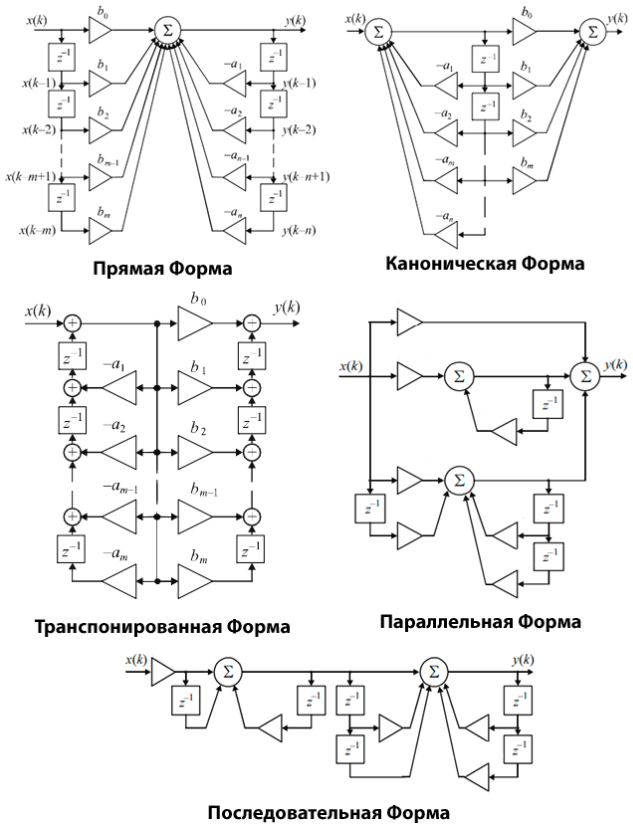


Рисунок 2. Разные формы реализации дискретных фильтров

Каноническая и прямая формы являются функционально эквивалентными. Однако при их практической реализации возникает ряд особенностей, которые необходимо учитывать. В канонической реализации используется общая линия задержки, что позволяет сократить количество требуемых ячеек, но увеличивает разрядность в промежуточных вычислениях увеличивая битовую ширину. Это снижает устойчивость

фильтра.

Для высоких частот **транспонированная форма** более устойчива по сравнению с прямой, потому что перемещение сумматоров к выходу снижает накопление ошибок, улучшая фазовую характеристику. На низких частотах применяется для улучшения устойчивости и снижения ошибок округления. Этот подход популярен для высокочастотных IIR-фильтров, где точность важна.

Для реализации фильтров высокого порядка с резкими изменениями в передаточной функции может быть целесообразно использовать **последовательную форму**, так как она позволяет разделить фильтр на каскады более низкого порядка. Это обеспечивает большую стабильность и точность при меньших требованиях к разрядности и снижает эффект накопления ошибок округления.

Параллельная форма часто требует больше аппаратных ресурсов, чем последовательная, так как каждый параллельный блок реализует свою часть фильтра, что увеличивает общую потребность в вычислительных элементах. Однако она обеспечивает меньшую временную задержку, поскольку позволяет выполнять вычисления быстрее за счет параллелизма.

Использование этих форм в зависимости от **точности, устойчивости и задержки** необходимо обосновывать требованиями к проекту, а не только выбором формы построения.

При проектировании и реализации БИХ-фильтра важно провести статического временного анализа (STA). Анализ позволит проверить, соответствует ли фильтр требованиям по временным задержкам и стабильности. Если в результате STA будут выявлены нарушения временных параметров, рекомендуется добавить триггеры (элементы задержки). Это поможет улучшить временные характеристики. Добавление задержек особенно важно для обеспечения синхронизации сигналов и повышения общей стабильности системы.

Функция передачи дискретного фильтра

Функция передачи дискретного фильтра H(z) описывает поведение фильтра в частотной области, то есть показывает, как входные сигналы преобразуются в выходные в зависимости от их частоты. Она позволяет исследовать особенности АЧХ и ФЧХ фильтра.

Если составить отношение Z-преобразования выходного сигнала к Z-преобразованию входного сигнала, получим функцию передачи:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{N} b_k z^{-k}}{1 + \sum_{k=1}^{M} a_k z^{-k}}$$

где:

- Числитель содержит коэффициенты b_k и описывает систему прямой связи, то есть ту часть, которая зависит непосредственно от входного сигнала.

- Знаменатель с коэффициентами a_k описывает обратную связь, или зависимость от предыдущих выходных значений.

Зная функцию передачи и коэффициенты, можем не только определить поведение фильтра, но и построить его в прямой, канонической и транспонированной форме.

Разложив числитель и знаменатель на множители, мы получим функцию передачи следущей формы:

$$H(z) = K \cdot \frac{(z - z_1)(z - z_2)\ldots(z - z_N)}{(z - p_1)(z - p_2)\ldots(z - p_M)}$$

где:

- z_1, z_2, ... z_N — нули функции передачи (корни числителя),

- p_1, p_2, ... p_M — полюса функции передачи (корни знаменателя),

- K — коэффициент усиления (b_0).

Полюса функции передачи — это значения, при которых знаменатель становится равен нулю. Они определяют устойчивость фильтра. Нули функции передачи — это значения, при которых числитель равен нулю. Нули определяют частоты, на которых фильтр подавляет или ослабляет сигнал.

В данной форме функции передачи можем судить об устойчивости и стабильности фильтра, а также построить его в последовательной форме. Однако, можно дальше упростить, разложив функцию переда на простые дроби:

$$H(z) = \sum_{i=1}^{M} \frac{r}{z - p_i}$$

где p_i и r_i — полюсы функции передачи и соответствующие им вычеты, с помощью которых можем судить о максимальной и минимальной временной задержке фильтра, также построить в параллельной форме.

Расчет коэффициентов фильтра

Удобным средством расчета фильтра является fdatool. fdatool — это инструмент в MATLAB, предназначенный для проектирования цифровых фильтров. Он предоставляет графический интерфейс, который позволяет пользователю быстро и удобно задавать параметры фильтра. fdatool автоматически генерирует коэффициенты фильтра, которые могут быть использованы в последующих вычислениях или для реализации фильтра в реальном времени.

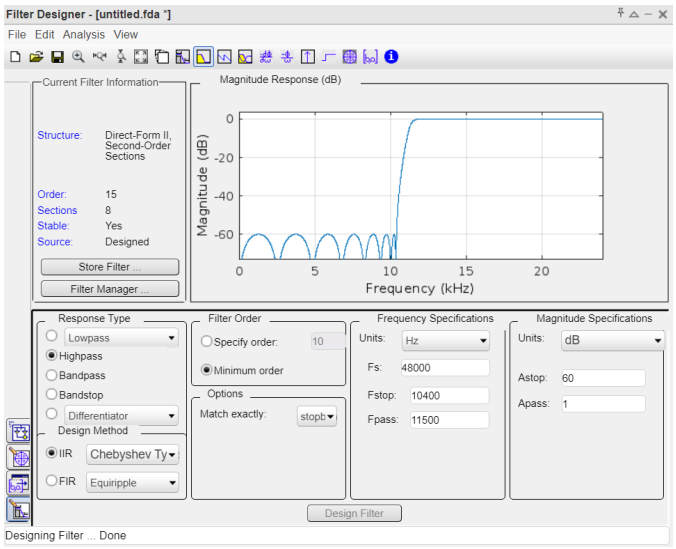


Рисунок 3. Окно инструмента fdatool

В области Response Type можно выбрать любой интересующий тип фильтра, Design Method – КИХ или БИХ фильтр. В **Frequency Specifications и Magnitude Specification** определяются частотные параметры характеристики фильтра и характеристики амплитудного отклика фильтра:

Fs – частота дискретизации,

Apass — неравномерность в полосе пропускания,

Astop — уровень затухания в полосе подавления,

Fpass — граничная частота полосы пропускания,

Fstop — граничная частота полосы затухания.

Вычислить коэффициенты фильтра можно программным путем. На пример: ФВЧ Чебышева 2 рода с пульсацией в полосе пропускания `0.5дБ` и полосе затухания

60 дБ, с граничной частотой полосе пропускания 6МГц и полосе затухания 5.2МГц. Частота дискретизации 20 МГц.

```
Fs = 20e6;              % Частота дискретизации
(Гц)
Wp = 6e6;               % Граничная частота
полосы пропускания (Гц)
Ws = 5.2e6;             % Граничная частота
полосы затухания (Гц)
Rp = 0.5;               % Пульсация в полосе
пропускания (дБ)
Rs = 60;                % Затухание в полосе
подавления (дБ)

Wp = Wp / (Fs / 2);     % Нормированная частота
пропускания
Ws = Ws / (Fs / 2);     % Нормированная частота
затухания

% Определение порядка фильтра
[n, Wn] = cheb2ord(Wp, Ws, Rp, Rs);

% Расчет коэффициентов фильтра Чебышева 2-го
рода
[b, a] = cheby2(n, Rs, Wn, 'high');

% Построение АЧХ и ФЧХ
freqz(b, a, 1024, Fs);
% Построение полюсов и нулей на комплексной
плоскости
zplane(b, a);

% Отображение коэффициентов
disp('Коэффициенты числителя (b):');
disp(b);
disp('Коэффициенты знаменателя (a):');
disp(a);
```

Использование SystemRDL для проектирования регистровых блоков

Артем Кашканов

Telegram: @radiolok

Обсуждение и комментарии: link

Аннотация

В статье рассматривается возможности языка SystemRDL для описания регистров состояния и управления с последующим получением как исходных кодов будущего IP-блока, так и документации на него.

Ключевые слова: SystemVerilog, SystemRDL, PeakRDL, CSR

SystemRDL

В процессе разработки цифрового устройства, перед разработчиком встает сразу несколько задач:

Во-первых, написание непосредственного функционала устройства согласно спецификации.

Во-вторых, как правило IP-блок работает в составе более сложной системы и общение происходит посредством одной из стандартных шин - AXI, APB и других - требуется реализация интерфейса доступа к управляющим регистрам, включая сложные поведения, битовые поля и вот это вот все.

В-третьих, неплохо бы на эти самые регистровые поля получить соответствующую документацию. Или наоборот – из готовой документации получить регистровое поле.

И если решение первого пункта полностью отдано на откуп разработчику, то вместо реализации и второго и третьего пунктов можно сделать только один – описать регистровое поле с помощью языка SystemRDL.

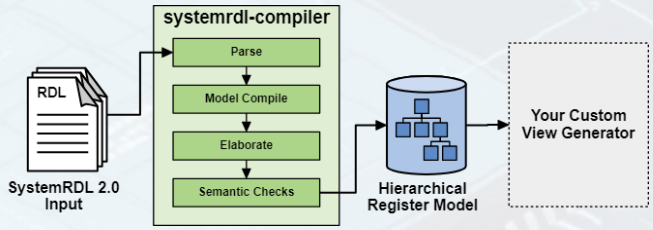


Рисунок 1 Структурная схема работы с systemRDL файлом

SystemRDL - это язык, поддерживаемый консорциумом Accellera, который был специально разработан для описания и реализации широкого спектра регистров состояния и управления. Используя SystemRDL, разработчики могут автоматически генерировать и синхронизировать представления регистров для спецификации, проектирования оборудования, разработки программного обеспечения, проверки и документирования.

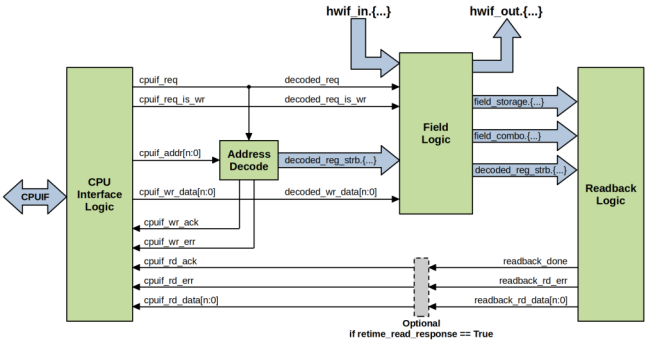
С помощью языка можно описать все регистровые поля, определить аппаратный и программный доступ к ним, добавить различные модификаторы и сигналы, а потом - преобразовать их и в SystemVerilog-файл, и в готовую документацию и в UVM сценарии и во многое другое. Не менее важно, что на SystemRDL могут писать люди, не знакомые с цифровой разработкой – что упрощает взаимодействие различных команд, например архитекторов, разработчиков железа и разработчиков софта.

Язык уже получил вторую версию спецификации, до сих пор обладает рядом недостатков, о которых я скажу позже, но все же позволяет ускорить процесс разработки сложных устройств.

PeakRDL

Однако одного языка нам недостаточно - нужен инструмент, который превратит описание регистров в, как минимум, подключаемый модуль на языке SystemVerilog. Существует несколько инструментов для работы с SystemRDL, как коммерческие - типа Agnisys, Semifore's CSR Compiler или Magillem, так и openSource - например PeakRDL, ORDT, RgGen и другие

PeakRDL, на мой взгляд, самый интересный, так как позволяет писать собственные плагины, расширяющие базовый функционал компилятора



Устанавливается он из PyPi репозитория, с помощью команды

```
python3 -m pip install peakrdl
```

Синтаксис языка

Описание регистрового поля имеет иерархическую структуру - адресное пространство -> регистровый файл -> регистр -> поле в регистре

адресное пространство

Адресное пространство – это набор регистров, регистровых полей или вложенных адресных пространств, имеющий свой адрес и границы. В одном файле может быть несколько адресных пространств, но рекомендую иметь в файле строго одно, так как по умолчанию PeakRDL скомпилирует только самое первое:

Тул	Лицензия	HDL	C-заголовки	UVM	IP-XACT (IEEE1685)	Doc
Open Register Designer Tool	Apache 2.0	SystemVerilog Verilog	C, C++, Python	+	-	JSpec
PeakRDL	GPL3	SystemVerilog	C, C++	+	+	HTML
RgGen	GPL3	SystemVerilog Verilog VHDL Veryl	C	+	-	Markdown

```
addrmap map_name_t{
 name = "какое-то адресное пространство на сколько-то там регистров"
 //пропишем настройки по умолчанию:
 default regwidth = 8;//ширина регистра
 default accesswidth = 8; //ширина интерфейсной шины
 default sw = rw;//полный доступ по стороны интерфейса
 default hw = r;//со стороны устройства - доступ только на чтение
 regfile_t a{};//инстанс какого-то поля
 reg_t b{};//инстанс какого-то регистра
} map_name_inst @0x100;//базовый адрес
```

При этом стандарт позволяет как сразу описывать новый инстанс пространства/файла/регистра/поля, так и определить его тип и структуру, а потом сделать множественное инстанцирование. Синтаксис аналогичен C - название до фигурных скобок работает как typedef, а после скобок - как имя инстанса.

регистровый файл

Регистровый файл - это логическая группировка нескольких регистров, которому можно дать свои свойства и, самое главное, определенное место в адресном пространстве. Например, если в устройстве имеется некоторое количество одинаковых подблоков и нужно обеспечить выровненный доступ к ним. Для этого один раз определяем регистровый файл, а потом - несколько раз его инстанцируем, указывая необходимый адрес, или выравнивание

```
regfile new_regfile_t{
    name = "регистровый файл подблока"
  //предположим, что структура этих регистров описана ранее
    reg1_t a;
  reg2_t b;
  };
  new_regfile_t block1 @0x10; //инстанцируем файл по адресу 0x10
  new_regfile_t block2 += 0x20; //еще одну копию - со сдвигом на 0x20 к предыдущему
  new_regfile_t block2[5] + 0x30;//и еще пяток копий со сдвигом на 0x30 к каждому и предыдущему
```

регистр и его поля

поле - самая низкоуровневая структура в языке SystemRDL. К полю осуществляется доступ со стороны устройства. Полем может являться как триггер, так и просто провод.

регистром - называется набор из одного или нескольких полей, которые доступны через интерфейс по определенному адресу.

```
reg reg1_t{
 name = "какой-то регистр на пару полей"
 field {
  name = "какое-то однобитовое поле"
 } a = 0;//без четкой позиции в регистре, со значением по умолчанию
 field {
  name = "какое-то 4-х битовое поле"
 } b[4:1] = 0xf;// с заданной позицией в регистре и значением по умолчанию
 field {
  singlepulse; //после записи бит будет активен только один такт времени и автоматически сбросится
 }c;
 field {
  rclr;//а это поле будет очищено при чтении
 } d;
field {
  swacc;//а это поле предоставит сигнал о том что его «потрогали»
 } e;
};
reg reg2_t {
    regwidth = 16;//регистр большой, а шина была - 8 бит.
   buffer_reads = true;//обеспечим к нему атомарный доступ на чтение
   buffer_writes = true;//и на запись
}reg @0x10;
```

User-Defined Properties

buffer_reads и buffer_writes – это, по стандарту, так называемые User-Defined Properties (UDP). Каждый компилятор предоставляет свой набор свойств, расширяющий возможности генерации. Например у Agnisus более 400 UDP. С их помощью можно активировать переход через клоковые домены (Clock-Domain Crossing), агрегировать блоки, кастомизировать интерфейсы к стандартным

шинам, отключать блоки для уменьшения потребления, добавлять функции безопасности, например коды четности и исправления ошибок (ECC), проверки избыточности (CRC) и многое другое.

У PeakRDL этих параметров всего шесть:

- buffer_reads - При активации - регистр доступен для чтения через двойной буфер: Read-buffered Registers
- rbuffer_trigger - Определяет сигнал, по которому значение регистра будет сохранено Read-buffered Registers
- buffer_writes - Если активно - запись в регистр атомарна через двойной буфер Write-buffered Registers
- wbuffer_trigger - Определяет сигнал, по которому значение регистра будет записано в него Write-buffered Registers
- rd_swacc - Активирует строб, который подсвечивает операцию чтения регистра со стороны интерфейса Read/Write-specific swacc
- wr_swacc - Активирует строб, который подсвечивает операцию записи регистра со стороны интерфейса Read/Write-specific swacc.

Двойной буфер на чтение работает следующим образом (на примере 32 разрядного регистра и 16 разрядной шины):

Сначала мы читаем младшее слово регистра. Одновременно с этим регистровый файл кладет старшее слово в промежуточный триггер. Вместо явного обращения к младшему слову можно использовать и внешний сигнал, определив параметр rbuffer_trigger, например

```
status_reg status1;
status_reg status2;
status2->buffer_reads = true;
/*чтение регистра status1 со стороны
интерфейса вызовет сохранение значения
регистра status2 в промежуточном буфере*/
status2->rbuffer_trigger = status1;
```

Второй операцией считываем старшее слово. В такой последовательности получаем цельное значение нашего регистра на момент считывания младшего слова. Чтение в обратном порядке принесет какое-то старое значение старшего слова и актуальное значение младшего слова.

Буферизация на запись работает аналогично, однако запись должна вестись от старшего слова к младшему - именно запись в младшее слово инициирует процесс записи в регистр значения целиком.

Можно писать и свои собственные свойства как плагины для PeakRDL.

Память и другие опции

Вышеописанные конструкции позволяют написать довольно небольшие регистровые поля. Однако, на помощь может прийти директива memory

```
external mem fifo_mem {
  mementries = 1024;
  memwidth = 32;
};
```

Она создаст область памяти на 1024 слова, шириной в 32 бита каждая. Однако, она не создает саму память – лишь выделяет запрашиваемую область и выводит провода для подключения этой памяти во внешний интерфейс. Где такое может понадобиться? Допустим у вас в устройстве есть ПЗУ с сырыми данными, или что-то вроде этого. Директива mem позволят включить ее в общее пространство, без костылей типа мультиплексоров. Директива external, к слову, может применяться и для отдельных регистров – тогда для него не будет создаваться внутренняя логика управления, и все будет отдано целиком на откуп разработчику. А вот на документацию это никак не отражается – т.е. детали реализации

остаются скрыты от глаз технического писателя.

Компиляция

После того как регистровый файл написан, необходимо его скомпилировать в исходный код на Verilog и в документацию. PeakRDL со всем этим справляется из коробки.

Для изучения функционала рассмотрим готовый пример Register description of Atmel XMEGA AU's SPI controller из папки examples:

генерация SystemVerilog файла

сгенерируем готовые файлы в папке regblock и с плоским APB3 интерфейсом

```
peakrdl regblock atxmega_spi.rdl -o regblock/
--cpuif apb3-flat
```

В папке regblock найдем два файла:

atxmega_spi_pkg.sv, в котором содержится package, описывающий структуру внутреннего регистрового интерфейса. Его будет использовать наш блок.

atxmega_spi.sv - основной файл с описанием модуля, который мы и будем инстанцировать в наш код. Заголовок его выглядит так:

```
module atxmega_spi (
    input wire clk,
    input wire rst,

    input wire s_apb_psel,
    input wire s_apb_penable,
    input wire s_apb_pwrite,
    input wire [2:0] s_apb_paddr,
    input wire [7:0] s_apb_pwdata,
    output logic s_apb_pready,
    output logic [7:0] s_apb_prdata,
    output logic s_apb_pslverr,

    input atxmega_spi_pkg::atxmega_spi__in_t
hwif_in,
    output atx-
mega_spi_pkg::atxmega_spi__out_t hwif_out
);
```

apb провода подключаются к вышестоящему блоку, _pkg - к внутренней логике нашего блока. Делается это, например, так:

```
//текущее значение нашего регистра
hwif_out.my_reg[0].my_field.value
//значение, которое будет записано в регистр
в следующем такте
hwif_in.my_reg[0].my_field.next
```

генерация HTML документации

```
peakrdl html atxmega_spi.rdl -o html_dir/
```

После запуска браузера (только нужно разрешить запускать локальные файлы, или воспользоваться bat файлом в папке вывода) показывается главное окно с описанием адресного пространства – во взятом примере оно состоит из четырех отдельных регистров:

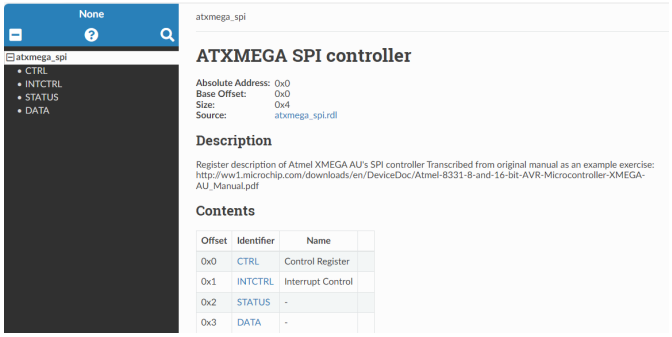


Рисунок 2 начальная страница документации

Если ткнуть на любой из них, то откроется описание по каждому из его полей:

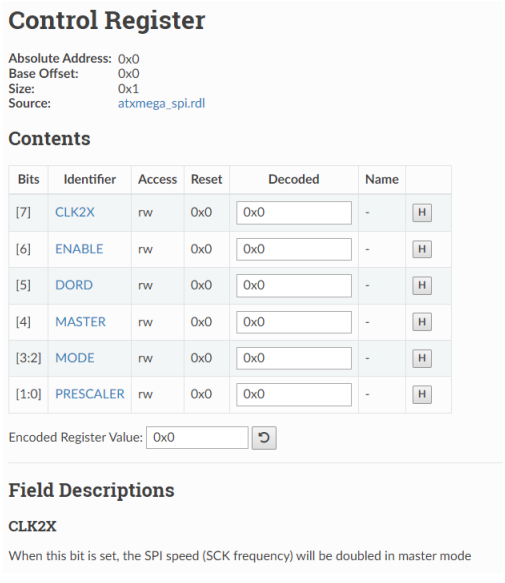


Рисунок 3 Описание полей регистра CTRL

Видны права доступа к этим полям, их значение после сброса, а также - есть поля ввода для значений отдельных полей, или регистра целиком. Это позволяет как набрать из отдельных полей итоговое значение регистра, так и разобрать имеющееся значение регистра в состояния его полей. Очень актуально для регистров со множеством символов.

Недостатки и ограничения

При всем достоинстве автоматического создания регистрового файла с реализованным интерфейсом связи с внешним миром, не обошлось и без недостатков.

На мой взгляд главный из них - параметризация. Вернее, ее отсутствие. Да, параметры есть, с иерархией, и при генерации можно их указать. Но в итоговом .sv файле все будет прибито гвоздями. Вы не получите параметризованный модуль на выходе – под каждую конфигурацию придется генерировать файл заново. Разумеется, волшебный sed-скрипт поможет вам автоматически допилить выходной файл до нужного состояния, но это, мягко скажем, тот еще костыль.

В документации помимо параметров есть еще и define - как Verilog-подобные, так и perl-вставки. Вот только они тоже применяются на этапе компиляции! Т.е. в языке есть две структуры для параметризации итогового блока и обе работают только на этапе компиляции. Я ничего не упускаю? С другой стороны быть может это ограничение только PeakRDL и другие тулы позволяют делать иначе? Стандарт на этот счет молчит как партизан.

Стоит также учитывать, что не все возможности SystemRDL поддержаны в PeakRDL.

Например, не поддерживается ключевое слово alias - когда вы можете сослаться на уже существующий регистр - таким образом у вас в одном адресном пространстве по разным адресам будет доступен один и тот же регистр.

Все адреса и отступы будут выравнены по ширине интерфейсной шины. Ширина шины доступа к регистрам определяется наибольшей accesswidth в группе. Исключением являются ситуации, когда регистр сам по себе меньше этой ширины.

```
//Максимальная ширина доступа accesswidth = 32, поэтому ширина шины данных 32 бит
reg {
    regwidth = 32;
    accesswidth = 32;
} reg_a @ 0x00; // OK. Обычный 32-разрядный регистр
reg {
    regwidth = 64;
    accesswidth = 32;
} reg_b @ 0x08; // OK. Широкий регистр на 64 бит, с доступом по 32 бита.
reg {
    regwidth = 8;
    accesswidth = 8;
} reg_c @ 0x10; // OK. Мелкий регистр, но адрес выравнен к ширине шины
reg {
    regwidth = 32;
    accesswidth = 8;
} bad_reg @ 0x14; // NOT OK. регистр влезает в шину целиком, а accesswidth - мелкий. Так нельзя
```

Выводы

Не смотря на свои ограничения, регистровый файл, созданный с помощью SystemRDL освобождает разработчика от головной боли по работе с внешним интерфейсом и по документации на этот файл. Все это вкупе дает единую точку работы с регистрами и снижает вероятность сопутствующих ошибок.

Полезные ссылки

- https://www.accellera.org/downloads/standards/systemrdl
- https://github.com/SystemRDL
- https://en.wikipedia.org/wiki/SystemRDL
- https://www.accellera.org/images/downloads/standards/systemrdl/SystemRDL_2.0_Jan2018.pdf

Процессор для Tang Nano 9K

Гуров В.В.

va1ery@ya.ru

Обсуждение и комментарии: link

Разработанная компанией Sipeed отладочная плата Tang Nano 9K была представлена в предыдущем номере журнала [1]. Теперь же хотелось бы по возможности вкратце изложить описанный в материалах компании Lushay Labs процесс создания ядра ЦП общего назначения [2]. В оригинальном материале читателю предлагается выстроить архитектуру набора команд, ядро ЦП и ассемблер. Мы же, дабы сократить объем, не будем рассматривать модуль верхнего уровня, ассемблер и та часть кода ядра, которая отвечает за вывод текста на OLED-дисплей (выделена желтым ниже). Мы вернемся к примеру со светодиодным счетчиком, но на этот раз реализуем его не аппаратно, а программно.

Под процессором обычно подразумевают устройство, способное читать и выполнять программы, состоящие из отдельных команд. Иначе говоря, процессор реализует заданную архитектуру набора команд—инструкций (instruction set architecture, ISA). Для каждой команды определяется своя последовательность нулей и единиц, именуемая кодом команды (opcode, сокр. от operation code - код операции, также известен как машинный код инструкции, instruction machine code, строка операции, opstring и т.д.), и процессор должен уметь декодировать эту последовательность, чтобы распознавать команды.

Предположим, инструкция "очистить регистр A" (в регистрах временно хранятся данные) представляется процессору как 01100001, — это и есть байт opcode. Но вместо того чтобы записать команду именно так, программист пишет ее на языке ассемблера, например CLR A, т.е. в текстовом представлении, которое ассемблер преобразует в вышеприведенный код операции. Обычно люди не пишут программы даже для ассемблера, а используют языки еще более высокого уровня абстракции, такие как C или Java для написания более краткого кода, который затем компилируется в код ассемблера и собирается им в двоичный код.

ISA

Чтобы создать минимальный набор команд, с помощью которого можно будет писать программы, требуется следующее:

- Способ работы с данными (загрузка данных и базовая арифметика)
- Способ сохранения результатов вычислений (обычно — в регистрах общего назначения)
- Способ получения вводимых пользователем данных
- Способ вывода данных пользователю
- Способ проверки условия для реализации выбора той или иной ветви вычислений по типу if-else
- Способ перехода из одного места программы в другое для организации циклических вычислений

Далее на языке Verilog мы напишем простой 8-битный процессор, это означает, что его регистры и операции будут 8-битными, а команды будут использовать 8-битные параметры.

В процессоре будет 4 регистра общего назначения, при этом условимся, что главный регистр называется AC, или аккумулятор (ACcumulator), и в нем будет по умолчанию сохраняться результат выполнения *любой* операции. Другие регистры просто получат букву в качестве обозначения, соответственно A, B и C.

Реализуем следующие операции:

- Очистить регистр
- Инвертировать регистр (т.е. заменить нули единицами и наоборот)
- Сложить число с содержимым регистра и результат поместить в тот же регистр
- Сложить 2 регистра вместе

Этот набор позволит выполнять сложение и умножение (поскольку умножение может быть реализовано как повторное сложение), а с помощью команды инвертирования — вычитание с использованием дополнительного кода, а также деление как повторное вычитание.

Реализовав эти четыре, мы получаем базовые арифметические операции. Помимо них понадобятся и некоторые другие, а именно:

- Сохранение содержимого главного регистра AC в одном из других регистров
- Установка значения для светодиодов
- Способ проверки нажатия кнопки пользователем
- Способ условного перехода между строками кода
- Команда ожидать X миллисекунд
- Способ остановки выполнения кода

Все это взятое вместе дает базовый набор команд для создания программ. Мы реализуем эти команды в виде восьми инструкций, где каждая инструкция может иметь один из четырех вариантов.

; CLR A/B/BTN/AC

CLR A ; очистить регистр A

CLR B ; очистить регистр B

CLR BTN ; при нажатии кнопки очистить регистр аккумулятора AC

CLR AC ; очистить регистр аккумулятора AC

; STA A/B/C/LED

STA A ; поместить содержимое аккумулятора AC в регистр A

STA B ; поместить содержимое аккумулятора AC в регистр B

STA C ; поместить содержимое аккумулятора AC в регистр C

STA LED ; установить значения элементов LED (светодиоды) по младшим шести битам регистра AC

; INV A/B/C/AC

INV A ; побитово инвертировать содержимое регистра a

INV B ; побитово инвертировать содержимое регистра b

INV C ; побитово инвертировать содержимое регистра c

INV AC ; побитово инвертировать содержимое аккумулятора

; HLT

HLT ; halt execution (stop program)

; ADD A/B/C/Constant

ADD A ; ac = ac + a

ADD B ; ac = ac + b

ADD C ; ac = ac + c

ADD 20 ; ac = ac + 20

; PRNT A/B/C/Constant (ac should have the screen char index)

PRNT A ; screen[ac] = a (a should be ascii value)

PRNT B ; screen[ac] = b (b should be ascii value)

PRNT C ; screen[ac] = c (c should be ascii value)

PRNT 110 ; screen[ac] = 110

; JMPZ A/B/C/Constant

JMPZ A ; go to line a in code if ac == 0

JMPZ B ; go to line b in code if ac == 0

JMPZ C ; go to line c in code if ac == 0

JMPZ 20 ; go to line 20 in code if ac == 0

; WAIT A/B/C/Constant

WAIT A ; wait a milliseconds

WAIT B ; wait b milliseconds

WAIT C ; wait c milliseconds

WAIT 100 ; wait 100 milliseconds

Итак, у нас есть 8 инструкций, большинство из которых имеют 4 варианта.

Чтобы различать инструкции, их нужно пронумеровать, как минимум, тремя битами, затем для каждой из них обозначить 4 варианта. Поскольку мы не будем использовать меньше байта на каждую инструкцию, предлагается следующая схема:

| 8 | **7** | **6** | **5** | 4 | 3 | 2 | 1 |

Здесь биты 1-4 обозначают вариант, биты 5-7 обозначают команду, восьмой бит содержит признак наличия параметра-константы. Так, например, если первая команда — CLR (см. выше), то ее четыре варианта будут такими:

CLR A ; 00001000 -> 0 000 1000

CLR B ; 00000100 -> 0 000 0100

CLR BTN ; 00000010 -> 0 000 0010

CLR AC ; 00000001 -> 0 000 0001

Далее рассмотрим следующую инструкцию ADD, здесь есть постоянный параметр, что отражено в нижней строке:

ADD A ; 00011000 -> 0 001 1000

ADD B ; 00010100 -> 0 001 0100

ADD C ; 00010010 -> 0 001 0010

ADD 20 ; 10010001 -> 1 001 0001

Первые три варианта точно такие же, как у команды CLR, в последнем же варианте, поскольку он использует постоянный параметр, установлен восьмой бит.

Поскольку каждый из 4 вариантов представлен своим уникальным битом, и только 1 бит для варианта всегда установлен, а также поскольку команда с постоянным параметром имеет свой бит признака константы, можно различать (декодировать) все эти варианты, проверяя всего один бит.

Отметим, что мы на самом деле еще нигде не указали само значение константы, — выше мы просто записали байт opcode, представляющий инструкцию «добавить константу», поэтому потребуется еще один байт с фактическим значением константы. Мы лишь помечаем эту команду дополнительным битом-флажком, чтобы знать, когда нужно для нее загрузить еще один байт со значением константы, а когда нет.

Спланировав набор инструкций, давайте рассмотрим теорию его реализации.

Ядро

Процессорное ядро, обрабатывающее инструкции, обычно выполняется в виде конструкции—конвейера (Pipeline). В нашем процессоре у нас будут следующие этапы конвейера:

- Fetch — загрузка очередной команды из памяти
- Decode — обработка/подготовка к вводу параметров команды
- Retrieve — необязательное извлечение из памяти другого байта (только, если используется постоянный параметр)
- Execute — запуск команды на выполнение после загрузки всего, что ей требуется

Для большинства процессоров это типичные этапы, и современные процессоры выполняют их параллельно: пока выполняется строка 1, извлекается строка 3, декодируется строка 2 и т.д. Мы не будем этого делать, так как это повышает сложность. Для простоты мы будем выполнять только один шаг за раз.

Рассмотрим, как реализуются эти этапы:

⇒*Этап выборки — Fetch*

Регистр, хранящий адрес нашего текущего местоположения в программе, обычно называют PC (program counter, перевод на русский «счетчик команд» не слишком удачен, поскольку в нем всего лишь хранится номер строки; есть еще вариант: Instruction Pointer, IP — указатель команды, но и эта аббревиатура слишком часто интерпретируется иначе).

На этапе выборки мы запросим байт по адресу, на который указывает регистр PC, этот байт, который мы получим, и является байтом opcode.

⇒*Этап декодирования — Decode*

Здесь мы обработаем считанный нами на этапе Fetch байт opcode. Проверим, нужно ли нам переходить к этапу извлечения (Retrieve), — это зависит от того, содержит ли единицу бит 8 «Имеется постоянный параметр».

В случае, если параметр не содержится в следующем байте, он подготавливается исходя из варианта.

⇒*Этап извлечения — Retrieve*

Если требуется считывание еще одного байта, то на этом этапе мы запросим этот следующий байт из памяти и сохраним его значение как параметр для текущей инструкции.

⇒*Этап выполнения — Execute*

Теперь у нас есть все, что требуется, и мы фактически выполняем нужную операцию.

Реализовав эти четыре этапа, мы получим

процессор, способный выполнять наш набор инструкций. Закончив с теорией, давайте, наконец, непосредственно перейдем к реализации.

Реализация

Для начала давайте создадим файл cpu.v со следующим определением одноименного модуля:

```
module cpu(
    input clk,
    output reg [10:0] flashReadAddr = 0,
    input [7:0] flashByteRead,
    output reg enableFlash = 0,
    input flashDataReady,
    output reg [5:0] leds = 6'b111111,
    input reset,
    input btn
);
//  ...
// ——> сюда будем добавлять код! <——
//  ...
endmodule
```

Первый вход — это тактовый сигнал, далее следуют 4 порта, необходимые для управления flash-памятью. Для взаимодействия с ней нам нужно в flashReadAddr установить адрес, по которому мы хотим прочитать данные, затем установить enableFlash на высокий уровень ("1"), чтобы начать процесс чтения. Затем нам нужно дождаться, пока flashDataReady не станет высоким, что означает, что байт был прочитан, тогда мы можем взять его значение из flashByteRead.

Также у нас имеется выходной регистр для управления светодиодами на плате, мы инициализируем этот регистр всеми единицами, поскольку наши светодиоды активны (горят) на низком уровне ("0"), это отключит их все по умолчанию.

Наконец, у нас есть два порта для кнопок, кнопка сброса перезапустит процессор, перезапустив код с нулевой строки, а вторая кнопка, называемая btn, используется командой CLR BTN.

Добавим некоторые определения (localparam):

```
localparam CMD_CLR = 0;
localparam CMD_ADD = 1;
localparam CMD_STA = 2;
localparam CMD_INV = 3;
//localparam CMD_PRNT = 4; оставить
закомментированным!!!
localparam CMD_JMPZ = 5;
localparam CMD_WAIT = 6;/
localparam CMD_HLT = 7;
```

Они определяют номера для каждой из наших 8 команд. Как мы говорили в разделе ISA, 3 бита будут определять, какая из 8 инструкций должна быть выбрана. Эти 3 бита и будут представлять одно из этих 8 определений localparam.

Далее нам понадобятся регистры:

```
reg [5:0] state = 0;
reg [10:0] pc = 0;
reg [7:0] a = 0, b = 0, c = 0, ac =
8'b00000000;
reg [7:0] param = 0, command = 0;
reg [15:0] waitCounter = 0;
```

Во-первых, у нас есть регистр конечного автомата state, который реализует конвейер выполнения, о котором мы говорили выше. Во-вторых, у нас есть регистр для счетчика команд pc, который хранит, на какой строке мы сейчас находимся в коде/памяти, мы начинаем с адреса 0 флэш-памяти.

Далее у нас есть четыре основных регистра, используемых в нашем ISA: A, B, C и AC, каждый из которых имеет длину 8 бит.

Вдобавок имеются еще два регистра, один из которых хранит текущую команду, а другой — текущий параметр. Так, например, если наша команда — ADD C, то она будет

сохранена в регистре command, а значение регистра с будет сохранено в param, а если текущая инструкция имеет постоянный параметр, то вместо с в param будет сохранен постоянный параметр.

Наконец, у нас есть регистр для команды WAIT. Эта команда должна ждать x миллисекунд, при 27 МГц, каждая миллисекунда составляет 27 000 тактов, поэтому у нас предусмотрен 16-битный регистр для подсчета 27 000 тактов, чтобы знать, что мы ждали 1 миллисекунду.

Теперь определим состояния конечного автомата нашего процессора:

```
localparam STATE_FETCH = 0;
localparam STATE_FETCH_WAIT_START = 1;
localparam STATE_FETCH_WAIT_DONE = 2;
localparam STATE_DECODE = 3;
localparam STATE_RETRIEVE = 4;
localparam STATE_RETRIEVE_WAIT_START = 5;
localparam STATE_RETRIEVE_WAIT_DONE = 6;
localparam STATE_EXECUTE = 7;
localparam STATE_HALT = 8;
localparam STATE_WAIT = 9;
//localparam STATE_PRINT = 10; оставить закомментированным!!!
```

У нас есть состояния для наших 4 этапов конвейера: выборка, декодирование, извлечение и выполнение. Команды, которые взаимодействуют с флэш-памятью, такие как выборка и извлечение, имеют 3 состояния: одно для инициализации операции чтения, одно для ожидания начала операции чтения флэш-памяти и одно для сохранения результата после завершения операции.

Причина, по которой это делается в 3 шага вместо 1 или 2, заключается в том, чтобы устранить «дребезг флагов». Если мы немедленно проверим, высок ли флаг dataReady, мы можем случайно прочитать флаг dataReady предыдущей операции чтения и решить, что наши данные готовы. Вместо этого сначала дождавшись, пока флаг готовности данных станет низким, и только затем проверив, высок ли он, мы гарантируем, что он высок из нашей текущей операции.

Кроме того, предусмотрено специальное состояние для HALT, которое в просто останавливает работу процессора после выполнения инструкции HLT. Наконец, у нас есть специальные состояния для ожидания x миллисекунд.

Реализация конечного автомата

Для начала основной блок always должен позаботиться об условии сброса. Если нажата кнопка reset, он должен сбросить все переменные до их начальных значений:

```
always @(posedge clk) begin
    if (reset) begin
        pc <= 0;
        a <= 0;
        b <= 0;
        c <= 0;
        ac <= 0;
        command <= 0;
        param <= 0;
        state <= STATE_FETCH;
        enableFlash <= 0;
        leds <= 6'b111111;
    end
    else begin
        case(state)
        // <-- поместить сюда все состояния
        // (states)!!!
        endcase
    end
end
```

Мы делаем сброс приоритетным, помещая весь конечный автомат в ветвь else.

Первое состояние — «Выборка», в котором нам нужно загрузить байт по адресу, указанному в регистре счетчика команд:

```
STATE_FETCH: begin
    if (~enableFlash) begin
        flashReadAddr <= pc;
        enableFlash <= 1;
        state <= STATE_FETCH_WAIT_START;
    end
end
```

```
STATE_FETCH_WAIT_START: begin
    if (~flashDataReady) begin
        state <= STATE_FETCH_WAIT_DONE;
    end
end
STATE_FETCH_WAIT_DONE: begin
    if (flashDataReady) begin
        command <= flashByteRead;
        enableFlash <= 0;
        state <= STATE_DECODE;
    end
end
```

Эти три состояния взаимодействуют с флэш-памятью для чтения байта. Первое состояние STATE_FETCH устанавливает нужный адрес по счетчику команд, высокий уровень на выводе enableFlash, затем мы переходим в состояние STATE_FETCH_WAIT_START, в котором ждем начала операции чтения.

В состоянии STATE_FETCH_WAIT_START мы просто ждем, пока флаг готовности flashDataReady не перейдет в низкий уровень, опять же, это делается дабы убедиться, что мы случайно не считываем статус флага из предыдущей операции по ошибке.

Заключительный этап STATE_FETCH_WAIT_DONE ждет, пока флаг готовности данных не перейдет в высокий уровень, где мы затем можем сохранить считанный байт кода операции в command и отключить флэш-память до тех пор, пока она нам снова не понадобится.

```
STATE_DECODE: begin
    pc <= pc + 1;
    // command has constant parameter
    if (command[7]) begin
        state <= STATE_RETRIEVE;
    end else begin
        param <= command[3] ? a : command
[2] ? b : command[1] ? c : ac;
        state <= STATE_EXECUTE;
    end
end
```

Следующий этап конвейера — это этап декодирования, где мы сначала увеличиваем счетчик команд на единицу, поскольку уже прочитали текущий код операции. Затем мы проверяем, имеет ли текущая команда постоянный параметр, что потребует чтения дополнительного байта из флэш-памяти, либо параметр является одним из четырех основных регистров.

Если вы помните из нашей архитектуры ISA, мы устанавливаем 8-й бит (при нумерации индекса от нуля до семи индекс восьмого бита равен 7) высоким в том случае, если текущая инструкция требует загрузки параметра—константы. Это легко проверить, и в этом случае мы перейдем к этапу его извлечения. В случае же, если команда не имеет такого параметра, мы сохраняем в param значение соответствующего регистра в зависимости от того, какой из четырех битов установлен в поле варианта текущей инструкции.

Таким образом, ADD A будет иметь установленный бит с индексом 3, тогда как ADD B будет иметь установленный бит с индексом 2. Не все команды имеют параметры, как, например, HLT, некоторые же команды имеют другие параметры, как, например, STA LED, где параметром является регистр массива светодиодов. Но, как бы там ни было, никогда не помешает сохранить один из 4 регистров в param, поэтому проще просто делать это всегда, а не только при необходимости.

Следующие 3 состояния предназначены для этапа извлечения. Они почти идентичны инструкциям выборки, за исключением того, что они делают, когда байт был прочитан:

```
STATE_RETRIEVE: begin
    if (~enableFlash) begin
        flashReadAddr <= pc;
        enableFlash <= 1;
        state <= STATE_RETRIEVE_WAIT_START;
    end
end
```

```
  STATE_RETRIEVE_WAIT_START: begin
     if (~flashDataReady) begin
         state <= STATE_RETRIEVE_WAIT_DONE;
     end
  end
  STATE_RETRIEVE_WAIT_DONE: begin
      if (flashDataReady) begin
          param <= flashByteRead;
          enableFlash <= 0;
    state <= STATE_EXECUTE;
          pc <= pc + 1;
      end
end
```

Первые два состояния точно такие же, как на этапе выборки. В состоянии STATE_RETRIEVE_WAIT_DONE мы сохраняем считанный байт в param и переходим к этапу выполнения. Мы увеличиваем счетчик команд, так как считываем один байт и должны перейти к адресу байта, содержащего следующую инструкцию.

Теперь переходим к этапу, на который приходится большая часть сложной работы. На этом этапе мы фактически выполняем текущую инструкцию, поэтому начнем с общей схемы, а затем в нее добавим каждую команду по очереди:

```
STATE_EXECUTE: begin
    state <= STATE_FETCH;
    case (command[6:4])
  ...
    endcase
end
```

Вложенный оператор case проверяет 3 бита, показывающие, какой именно является наша текущая инструкция. Здесь используются определения команд localparam, заданные выше (см. «Реализация»).

Первая команда, которую мы реализуем, — это CLR:

```
CMD_CLR: begin
    if (command[0])
        ac <= 0;
    else if (command[1])
        ac <= btn ? 0 : (ac ? 1 : 0);
    else if (command[2])
        b <= 0;
    else if (command[3])
        a <= 0;
end
```

Здесь мы очищаем регистры, для чего «проходимся» по четырем младшим битам кода команды (с индексами 0, 1, 2, 3), которые сообщают, с каким вариантом мы работаем, и выполняем соответствующее действие. Большинство вариантов просто устанавливают регистр в ноль, за исключением CLR BTN, который очищает ac только если в данный момент нажата кнопка пользователя, в противном случае сохраняется текущее значение ac.

Следующая команда, которую мы реализуем, — ADD. Эта команда намного проще, поскольку значение всегда сохраняется в ac, а параметр уже сохранен в param на этапе извлечения.

```
CMD_ADD: begin
    ac <= ac + param;
end
```

Далее у нас есть команда STA, которая сохраняет регистр ac в регистр назначения на основе варианта:

```
CMD_STA: begin
    if (command[0])
        leds <= ~ac[5:0];
    else if (command[1])
        c <= ac;
    else if (command[2])
        b <= ac;
    else if (command[3])
        a <= ac;
end
```

Большинство из вариантов просто сохраняют регистр ac в другой регистр, опять же, поскольку значение в каждой из операций равно ac, сохранение его в param нам ничего не даст, поэтому нужно отработать здесь каждый из четырех

вариантов. Первый вариант инвертирует значение и берет только шесть младших бит, у нас всего 6 светодиодов.

Следующая инструкция — это инструкция INV, которая просто инвертирует биты одного из регистров:

```
CMD_INV: begin
    if (command[0])
        ac <= ~ac;
    else if (command[1])
        c <= ~c;
    else if (command[2])
        b <= ~b;
    else if (command[3])
        a <= ~a;
end
```

Здесь нечего особенно объяснять, каждый вариант обрабатывается так же, как и раньше, и просто «переворачивает» биты регистра.

Следующая команда — JMPZ, которая переходит на другую строку кода, если регистр ac в данный момент равен 0.

```
CMD_JMPZ: begin
    pc <= (ac == 8'd0) ? {3'b0,param} : pc;
end
```

Адрес, по которому мы хотим перейти, уже сохранен в param, поэтому здесь мы просто проверяем, равен ли ac нулю, в этом случае мы устанавливаем текущее значение pc на адрес, по которому мы хотим перейти. В противном случае мы сохраняем текущее значение pc, фактически ничего не делая.

Стоит отметить, что счетчик команд имеет длину 11 бит, а параметры имеют длину всего 8 бит, и это означает, что хотя теоретически программы могут быть длиной 2048 строк (поскольку счетчик программ увеличится до этого значения, прежде чем вернуться к нулю), инструкция перехода позволяет перейти только по адресу до строки 256. Это ограничение ISA, которое придется обходить при проектировании программ.

Следующая инструкция, которую нам нужно реализовать, — это инструкция WAIT, в которой мы ждем x миллисекунд, где x — это значение, сохраненное в param

```
CMD_WAIT: begin
    waitCounter <= 0;
    state <= STATE_WAIT;
end
```

Здесь мы переходим в состояние STATE_WAIT и ждем запуска следующей команды.

Последняя инструкция — это инструкция HLT, которая просто останавливает выполнение:

```
CMD_HLT: begin
    state <= STATE_HALT;
end
```

Здесь мы переходим в особое состояние, в котором мы просто ничего не будем делать, поскольку мы завершили программу.

Итак, мы завершили внутренний оператор case и реализовали все инструкции нашей ISA. Теперь мы можем вернуться к реализации конечных состояний во внешнем операторе case, которые являются этими особыми состояниями, добавленными нами для определенных инструкций.

Так, у нас есть состояние ожидания x миллисекунд:

```
STATE_WAIT: begin
    if (waitCounter == 27000) begin
        param <= param - 1;
        waitCounter <= 0;
        if (param == 0)
            state <= STATE_FETCH;
    end else
```

```
        waitCounter <= waitCounter + 1;
end
```

Здесь, мы отсчитываем 27 000 тактов, что равно 1 миллисекунде, а затем уменьшаем param. Так что если param был равен 20, мы сделаем это 20 раз, по сути, ожидая 20 миллисекунд, а затем, когда param станет 0, мы вернемся к нашему конвейеру.

Наконец, у нас есть состояние STATE_HALT, его можно включить, чтобы подчеркнуть, что мы ничего здесь не делаем.

```
STATE_HALT: begin
end
```

На этом наш модуль процессора завершен и теперь может выполнять программы, написанные с помощью нашего набора инструкций.

Тестбенч

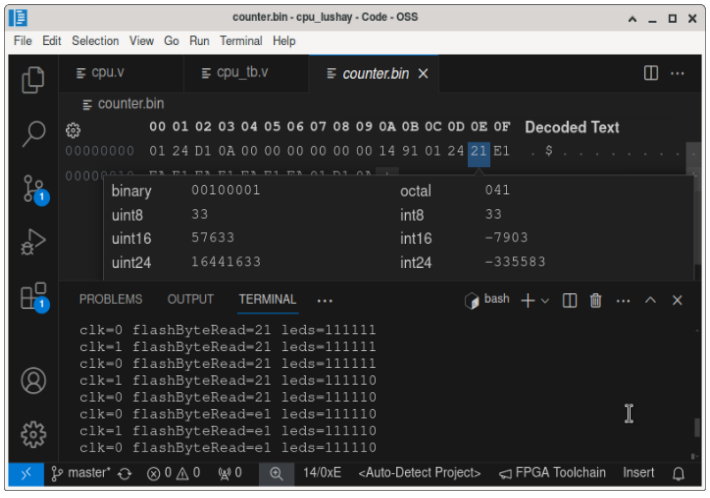
Как уже говорилось, в оригинальной статье [2] далее предлагается создать модуль верхнего уровня, файл ограничений, написать программу для ассемблера на основе разработанного выше набора команд, написать сам ассемблер, сгенерировать бинарный код и битстрим, все это вместе загрузить во флеш-память платы Tang Nano 9K и наблюдать за светодиодами, как в предыдущей статье [1]. Я же просто приведу здесь результаты тестирования самого модуля cpu.

Вот начало тестбенча:

```
`timescale 1ns/1ns
module cpu_tb;
  // Test inputs
   reg reset;
   reg clk;
   reg [7:0] flashByteRead;
   reg flashDataReady;
   reg btn;
  // Test outputs
   wire [10:0] flashReadAddr;
   wire enableFlash;
   wire [5:0] leds;
```

```
    cpu dut(
        clk,
        flashReadAddr,
        flashByteRead,
        enableFlash,
        flashDataReady,
        leds,
        reset,
        btn
    );
      initial begin
        clk = 0;
        #10;
        flashDataReady = 0 ;
        flashByteRead = 8'b00000000 ;
        reset = 0 ;
        btn = 0;
        #10 clk = 1;
        #10;
        flashDataReady = 0 ;
        flashByteRead = 8'b00000000 ;
        #10 clk = 0;
        #10;
        flashDataReady = 0 ;
        flashByteRead = 8'b00000001 ;
        #10 clk = 1;
        #10;
        flashDataReady = 0 ;
        flashByteRead = 8'b00000001 ;
        #10 clk = 0; #10;
                    ...
```

Далее в том же духе -- чередование установок и сбросов flashDataReady с помещением во flashByteRead строк бинарного кода программы счетчика counter.bin [3].

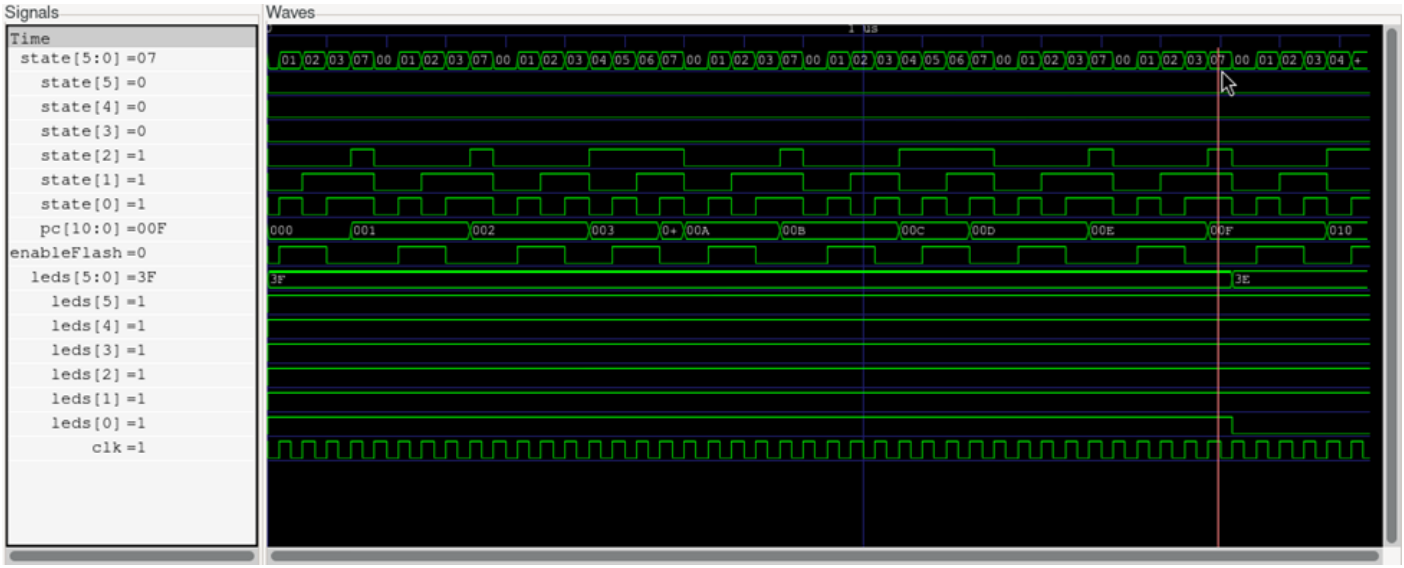


Последовательно вводя коды команд во flashByteRead, на 40-41 тактах достигаем

момента первого срабатывания счетчика. На приведенной ниже временной диаграмме этот момент обозначен линией визира, стрелка указывает на выход конечного автомата из состояния выполнения.

Источники:

1. Гуров В.В. Опенсорс для ПЛИС... и наоборот URL: https://t.me/fpgasystems_fsm/45
2. Tang Nano 9K: Our First CPU URL: https://learn.lushaylabs.com/tang-nano-9k-first-processor/
3. Tang Nano 9K Examples URL: https://github.com/lushaylabs/tangnano9k-series-examples/tree/master/cpu/programs



Вывод DVI с нуля под Yosys

Кудинов Максим

Telegram: @m_kudinov

Обсуждение и комментарии: link

Аннотация

Последовательные интерфейсы вывода изображения, такие как DVI и HDMI, требуют относительно высоких частот для сериализации, из-за чего работа с ними на ПЛИС сводится к инстанциированию пары готовых IP блоков от вендора.

Цель же этой статьи - посмотреть на устройство DVI на более низком уровне, по возможности полностью избегая готовые решения. Конечно, логика на LUT будет значительно медленнее аппаратных аналогов, но в то же время решение без сторонних IP получится более универсальным и дает возможность использовать открытые инструменты синтеза с минимальными изменениями кода под конкретную плату.

Введение

DVI был выбран по причине полной совместимости с HDMI, который присутствует на большом количестве отладочных плат. В моем случае будет использоваться плата Gowin Tang Primer 20K dock.

Использование ее с Yosys тулчейном стало возможно благодаря реверс-инжинирингу битстрима - этим занимается проект Apicula. Перед загрузкой прошивки на плату надо пройти несколько этапов в различных тулах: синтез под примитивы FPGA в Yosys (synth_gowin), размещение (PnR) в nextpnr (nextpnr_himbaechel), генерация битстрима для платы с помощью gowin_pack (Apicula), и, наконец, загрузка на плату через OpenFPGALoader.

Apicula все еще находится в активной разработке, так что далеко не все существующие примитивы от Gowin доступны, но в нашем случае это не важно.

Что такое DVI?

С точки зрения внешних управляющих сигналов DVI очень похож на VGA. У нас есть 2 счетчика, отвечающие за координаты X и Y. Перемещение по экрану происходит построчно, слева направо и сверху вниз.

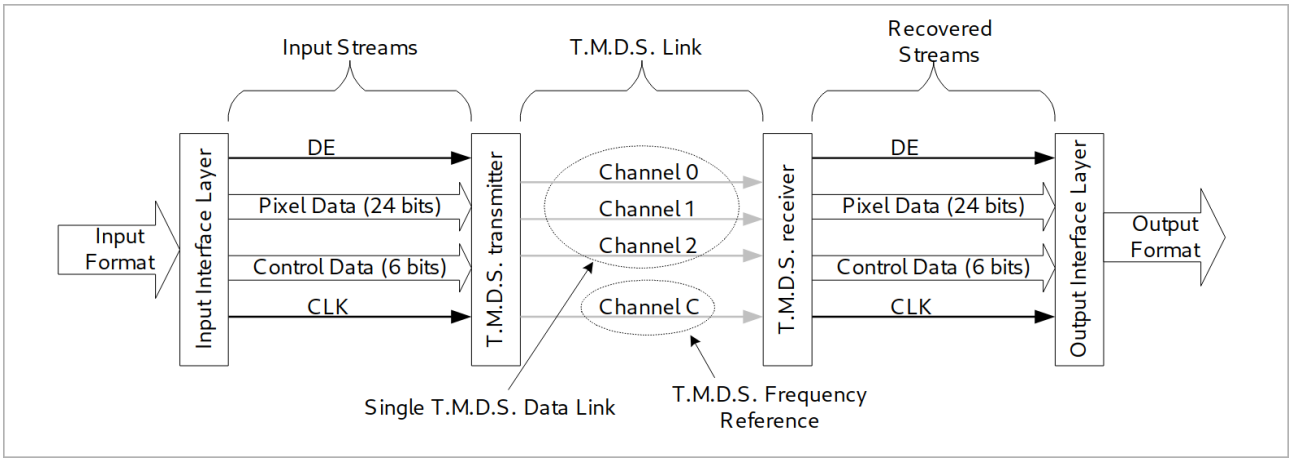
Между кадрами также есть пропуски (blanking), во время которых счетчики продолжают инкрементироваться, но изображение не выводится.

Мы будем реализовывать самое простое разрешение - 640x480, для чего pixel_clk должен быть 25.175 МГц.

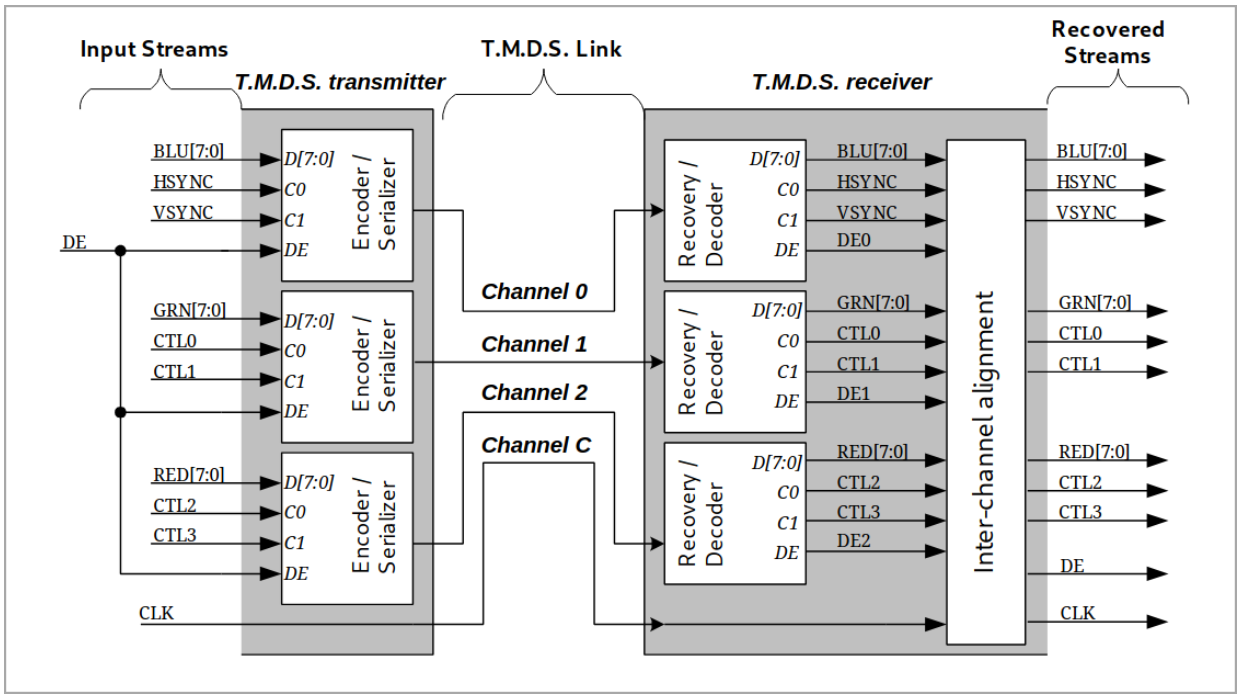
Но на этом сходства заканчиваются: VGA просто отправляет аналоговый сигнал по трём каналам цвета, а DVI интерфейс цифровой, в котором данные передаются последовательно по трём каналам, состоящим из дифференциальных пар.

Каждый из цветов RGB имеет разрядность 8 бит, так что до передачи данные сериализуются. Но передаются цвета не в изначальном формате, а в закодированном по протоколу T.M.D.S.

Transition minimized differential signaling обеспечивает минимизацию переходов битов сигнала из одного состояния в другое, что снижает генерацию помех. Также в алгоритме предусмотрен DC balancing, меняющий периодически полярность сигналов, чтобы предотвратить заряд кабеля.

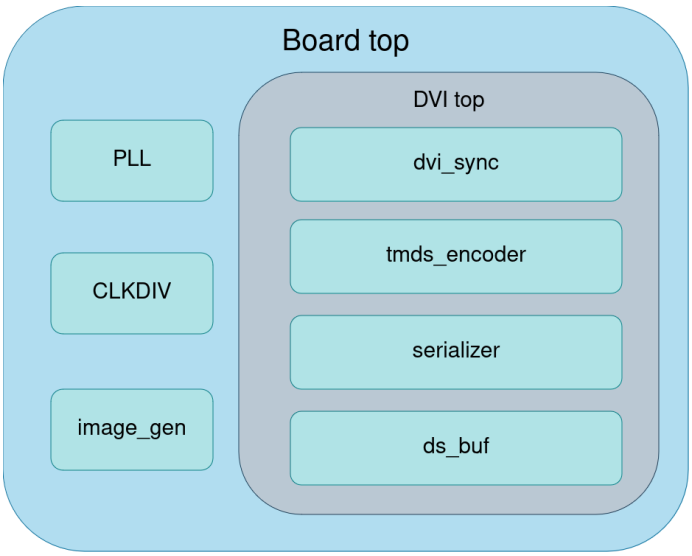


DVI интерфейс между передатчиком и приёмником



Каналы RGB с кодированием и сериализацией

Таким образом, задачу можно разбить на 3 части: генерация сигналов синхронизации (hsync/vsync), кодировка данных и сериализация.



TMDS Encoder

Генерация hsync/vsync ничем не отличается от VGA, так что ее мы пропустим, она была разобрана в моей статье про игру в Pong в предыдущем номере журнала.

Перейдем сразу к кодировке. В спецификации на DVI дана очень удобная картинка с алгоритмом, будем рассматривать ее по частям.

DE (Data Enable) - сигнал, обозначающий то, что мы находимся в видимой области экрана; D - сам цвет; C0/C1 - это hsync и vsync соответственно.

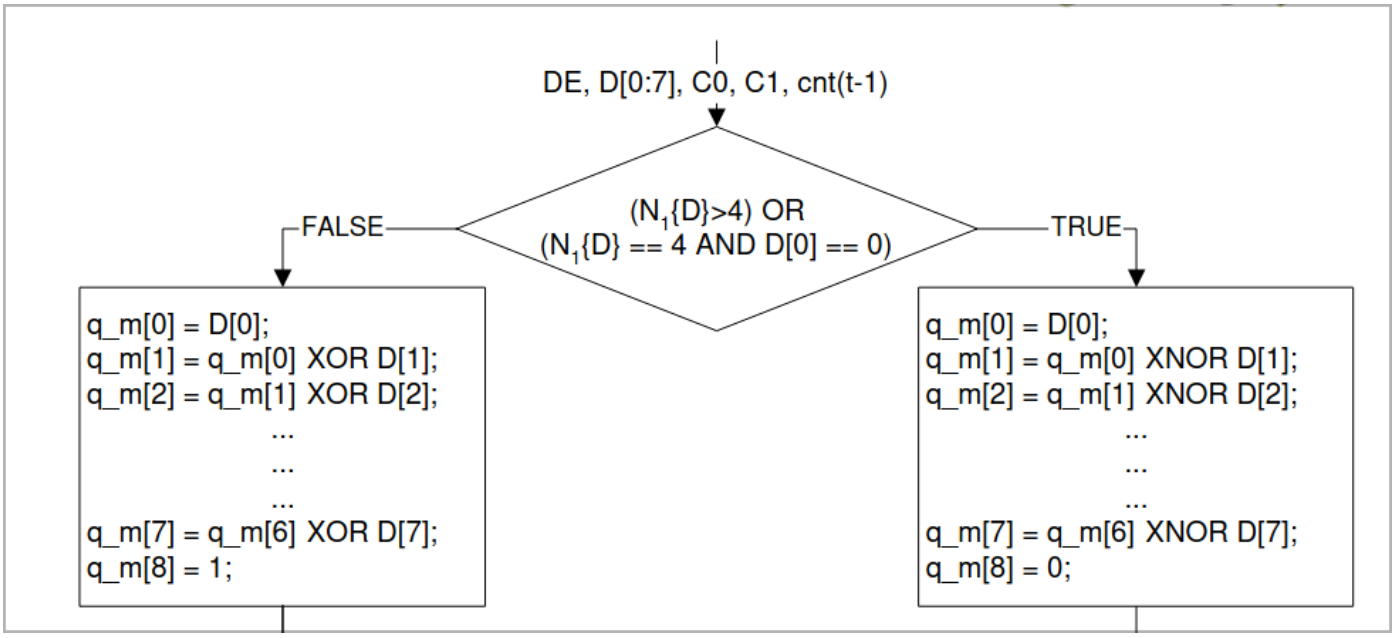
Cnt(t) отвечает за DC balancing - это знаковое число, его положительное значение обозначает “перевес” в пользу единиц в потоке, а отрицательное - в пользу нулей соответственно.

Cnt(t-1) - значение Cnt(t) с предыдущего такта.

$N_1\{x\}$, $N_0\{x\}$ - количество 1 или 0 в сигнале соответственно.

В TMDS к 8 битному сигналу добавляется еще 2 бита - флаг минимизации и флаг инверсии, на выходе получаем 10 бит.

Первым делом минимизируем переходы: если есть “перевес” в пользу единиц - то выполняем XNOR между разрядами цвета, а если нет - то XOR.

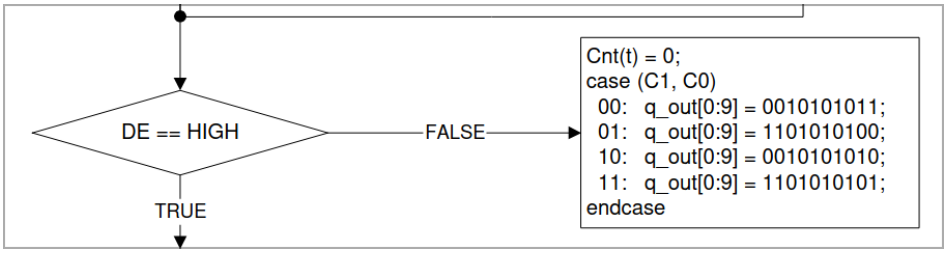


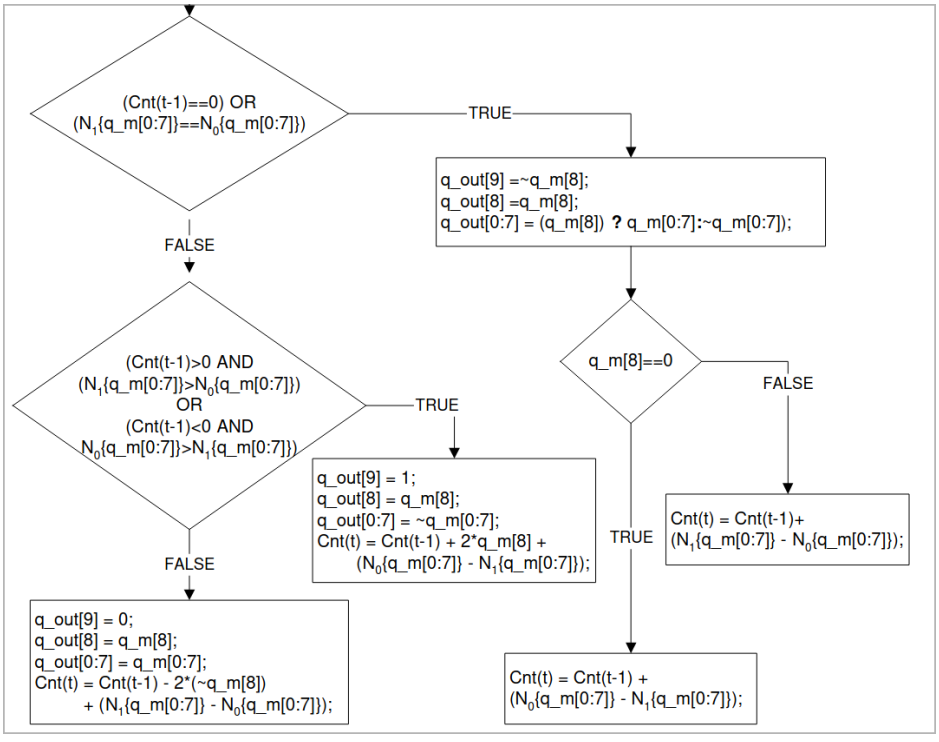
```
always_comb begin
      N1D =  4'(D[0]) + 4'(D[1]) + 4'(D[2])
            + 4'(D[3]) + 4'(D[4]) + 4'(D[5])
            + 4'(D[6]) + 4'(D[7]);

      q_m[0] = D[0];

      if ((N1D > 4) ||
            (N1D == 4 && ~D[0])) begin
            q_m[8] = 1'b0;

            for (int i = 0; i < 7; i++)
begin
            q_m[i + 1] = q_m[i]
                        ~^ D[i + 1];
            end

      end else begin
            q_m[8] = 1'b1;

            for (int i = 0; i < 7; i++)
begin
            q_m[i + 1] = q_m[i]
                        ^ D[i + 1];
            end
      end
```

Потом мы смотрим, находимся ли мы в видимой области экрана или нет. Если нет, то выводим кодировку на основе vsync/hsync. Таким образом, в отличии от VGA, значение цвета никогда не попадёт на приемник в запретной зоне:

```
cnt_next = '0;
```

```
case ({C1, C0})
  2'b00: q_next = 10'b1101010100;
  2'b01: q_next = 10'b0010101011;
  2'b10: q_next = 10'b0101010100;
  2'b11: q_next = 10'b1010101011;
endcase
```





Здесь довольно большой кусок логики для DC balancing, но суть сводится к тому, чтобы на основе предыдущего значения Cnt(t -1) принять решение, инвертировать ли биты цвета, или нет. А также сформировать новое значение Cnt(t).

Код довольно однообразный, так что покажу только отрывок для ветки false. Все целиком можно посмотреть на GitHub.

```
if (((cnt > 0) && (N1_qm > N0_qm)) ||
    ((cnt < 0) && (N0_qm > N1_qm))) begin

      q_next[9]   = 1'b1;
      q_next[8]   = q_m[8];
      q_next[7:0] = ~q_m[7:0];
      cnt_next     = cnt +
      (W_CNT'(q_m[8]) << 1) +
       W_CNT'(N0_qm) - W_CNT'(N1_qm);

end else begin
      q_next[9]   = 1'b0;
      q_next[8]   = q_m[8];
      q_next[7:0] = q_m[7:0];
      cnt_next     = cnt -
      (W_CNT'({~q_m[8]}) << 1) +
       W_CNT'(N1_qm) - W_CNT'(N0_qm);
end
```

На этом кодировка закончена, осталось только записать значения перед выходом в регистры, чтобы не было проблем с таймингом и глитчами.

```
      always_ff @(posedge clk_i) begin
      if (rst_i) begin
            cnt <= '0;
      end else begin
            cnt <= cnt_next;
      end
      end

      always_ff @(posedge clk_i) begin
        q_out <= q_next;
      end
```

Serializer

Для высокой частоты стоит использовать специальный примитив аппаратного сериализатора, но мы будем писать на логике. Сериализатор будет представлять из себя сдвиговый регистр, работающий на частоте в 10 раз превышающей pixel_clk, чтобы за один период смены цвета отправить на выход все 10 бит закодированного сигнала.

Получается, что в дизайне будет 2 тактовых сигнала. Tang Primer 20k имеет аппаратный генератор на частоте 27 МГц, что не подходит для желаемого pixel_clk (25.175 МГц).

Тут нет другого выбора, кроме как воспользоваться примитивом вендора для PLL, чтобы сгенерировать сигнал 252 МГц (достаточно близко к 25.175, чтобы не вызывать проблем), а потом поделить в 10 раз до 25.2 МГц.

Я пробовал реализацию делителя на счетчике, но даже с BUFG сигнал не заводился в глобальное клоковое дерево в nextpnr, что приводило к артефактам на экране. Так что делитель осуществляется через примитив CLKDIV.

После чего мы получаем pixel_clk - 25.2 МГц и serial_clk 252 МГц.

```
localparam CNT_MAX = DATA_W - 1;
assign load = cnt == CNT_MAX;

      always_ff @(posedge clk_i) begin
      if (rst_i) begin
            cnt <= '0;
      end else if (load) begin
            cnt <= '0;
      end else begin
            cnt <= cnt + 1'b1;
      end
      end

      always_ff @(posedge clk_i) begin
      if (rst_i) begin
            shift_reg <= '0;
      end else if (load) begin
            shift_reg <= data_i;
      end else begin
            shift_reg <= { 1'b0, shift_reg
[DATA_W-1:1] };
      end
      end
```

Когда мы отсчитали отправку 10 бит - берем новое значение со входа.

DS_buf

Ну и последнее - сделать дифференциальный сигнал. Здесь блок вендора показал себя особенно лучше в отчетах по трассировке pixel_clk, но на разрешении 640x480 формирование сигналов через LUT тоже работает без нареканий.

```
module ds_buf (
	input  logic clk_i,
	input  logic in,
	output logic out,
	output logic out_n
);

	always_ff @(posedge clk_i) begin
	out   <=   in;
	out_n <= ~ in;
	end

endmodule
```

Запуск OSS тулчейна

Я использую shell скрипт, который запускает все этапы сборки, давайте посмотрим на каждый из них.

Сначала указывается конфигурация платы:

```
device="GW2A-LV18PG256C8/I7"
board="tangprimer20k"
```

Потом перевожу весь исходный код в Verilog 2005, иначе Yosys выдает ошибку на import package:

```
yosys -p "read_verilog sv2v/converted.v;
synth_gowin -json design.json"
```

Непосредственно синтез на примитивы Gowin:

```
sv2v -I rtl/include/ -y rtl/ rtl/board_top.sv
--write=sv2v/converted.v
```

Трассировка (PnR):

```
nextpnr-himbaechel --json design.json --write
design_pnr.json  \
--device $device --vopt family=GW2A-18 --vopt
cst=pins.cst
```

Генерация битстрима:

```
gowin_pack -d $device -o pack.fs de-
sign_pnr.json
```

И загрузка в плату:

```
openFPGALoader -b $board pack.fs
```

И если все было сделано корректно, то на экране появится изображение.

Хотелось бы цветное, но, увы, даже 256x256 не поместилось в BRAM. Повод сделать приемник DVI.

Источники

https://www.fpga4fun.com/HDMI.html

https://www.fpga4fun.com/files/WP_TMDS.pdf

https://glenwing.github.io/docs/DVI-1.0.pdf

https://github.com/YosysHQ/apicula/tree/master/examples/himbaechel/DVI

https://github.com/hdl-util/hdmi

Удаленное программирование ПЛИС, с использованием программного пакета Xilinx ISE 14.7

Аноним

Обсуждение и комментарии: link

.

Используемое оборудование и ПО:

FPGA – Spartan 6 в составе отладочной платы SP605.

ПК сервер – компьютер с ОС Windows 7.

ПК клиент – компьютер с ОС Windows 7.

Используемое ПО – программный пакет ISE 14.7.

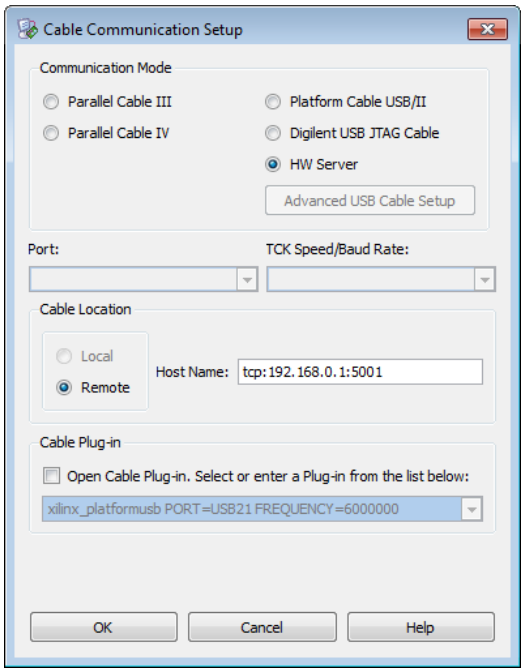
Решение:

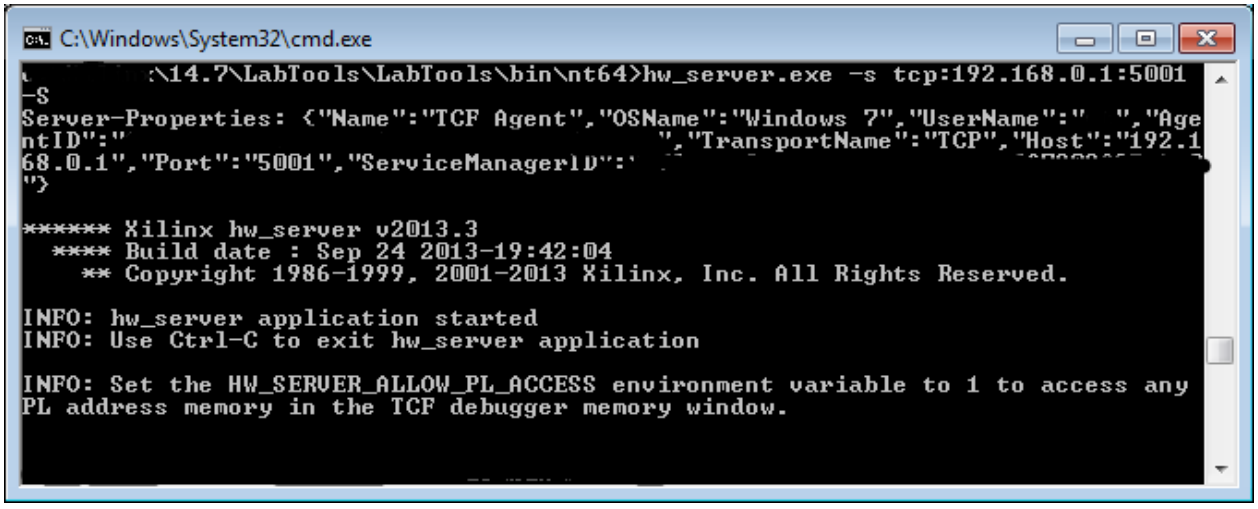
Для удаленного доступа к JTAG необходимо на ПК-сервере установить либо полный пакет ISE, либо ограничиться Lab-Tools. LabTools не включает средств разработки, а содержит только инструменты для программирования и отладки, которых вполне достаточно для наших целей. На ПК-клиенте, чаще всего ведётся разработка ПО, поэтому там разумно иметь полный пакет ISE. Допустимо устанавливать программы в виртуальной ОС, если она правильно настроена и видится в сети как обычный компьютер, то всё будет работать. Так же заранее следует настроить сеть, чтобы клиент и сервер находились в одной сети и имели доступ друг к другу.

На сервере необходимо запустить консольное приложение из *...\14.7 \LabTools\LabTools\bin\nt64 \unwrapped\hw_server.exe*. Запускать следует с ключом -s (не путайте ключи -s и -S, регистр имеет значение), который позволяет прямо указать протокол, интерфейс и порт, которые будут использоваться для доступа. По умолчанию используется TCP порт 3121 и

локальный интерфейс 127.0.0.1. (проверить эти параметры можно используя ключ -S). Доступ извне к этому интерфейсу затруднителен, а порт может быть занят другим приложением, поэтому следует указать конкретные значения, в которых вы уверены. В результате, консольная команда будет выглядеть следующим образом: *hw_server -s tcp:<ip компьютера>:<номер порта>*. Например: *hw_server -s tcp:192.168.0.1:5001*.

На стороне клиента запускается impact и открывается окно *Boundary Scan*. Далее идём в *Output-> Cable Setup*. Выбираем *HW Server* и в поле *Host Name* вводим сетевые параметры в том же формате что и на сервере: *tcp:<ip компьютера>:<номер порта>*. Например: *tcp:192.168.0.1:5001*. Нажимаем кнопку «OK» и все готово. Подтверждением успешного подключения является сообщение в консоли impact'а *INFO:iMPACT - Connection established.* После этого можно программировать ПЛИС или Flash так же, как если бы они были подключены к клиентскому компьютеру.





Заметки ПЛИСовода. Часть вторая.

Туровский Дмитрий Николаевич

E-mail: dim7881@rambler.ru

Обсуждение и комментарии: link

В прошлых заметках мы рассмотрели некоторые типичные ошибки и недочёты. Было бы ошибочно полагать, что на этом они заканчиваются. Рассмотрим ещё некоторые часто встречающиеся ошибки:

1. Отсутствие комментариев в коде.
2. Некорректные констрейны или их отсутствие.
3. Отсутствие пересинхронизации внешнего сигнала сброса.
4. Проектирование без этапа RTL-симуляции модулей.
5. Неиспользование внутренних средств отладки.

Разберёмся с каждым пунктом подробнее.

Комментарии

Отсутствие комментариев часто приводит к дополнительной трате времени на изучение и поиск необходимой части кода. В купе с некорректными названиями сигналов (wire'ов, reg'ов и прочего) это влечёт за собой ещё больше затрат. Можно придерживаться мнения, что понятный код в комментариях не нуждается, да, но в целом умение писать их кратко, информативно в нужных местах - полезный навык. В качестве примера ниже приводится код:

```
module reg_file
#(
    parameter XLEN = 32
)
(
  input    clk,      // input clock
  input    we3,      // write enable input
  input    [4:0] a1, // read address bus #1
  input    [4:0] a2, // read address bus #2
  input    [4:0] a3, // write address bus
  input    [XLEN-1:0] wd3,// write data bus
  output   [XLEN-1:0] rd1,// read data bus #1
  output   [XLEN-1:0] rd2 // read data bus #2
);
    // === Wire's, reg's and etc... ===
    reg [31:0] x[XLEN-1:0];   // Registers

    // === Assignments ===
    assign rd1 = (a1 != 0) ? x[a1] : 0;
    assign rd2 = (a2 != 0) ? x[a2] : 0;

    always @(posedge clk)
        if (we3)
            x[a3] <= wd3;
endmodule
```

Констрейны

От их упоминания может уже стать страшно. Автору становится ещё страшнее, если в проекте нет никаких констрейнов.

Очень важно, начиная с первых проектов, разобраться и приучить себя как минимум прописывать констрейн create_clock для системного синхросигнала. Если же не прописать этот базовый констрейн, то компиляция проекта будет происходить со значением по умолчанию, а не с реальным. Это потенциально может привести к непредсказуемому результату работы проекта после компиляции в ПЛИС.

Также встречается ситуация, когда констрейн для синхросигнала в проекте есть, но значение - некорректно. Например, к ПЛИС подключён источник синхросигнала частотой в 24 МГц, но в файле временных ограничений проекта констрейн прописан под 6 МГц. Получается, что констрейн есть, но значение отличается от реального.

Подытожим:

1. Прописываем констрейн create_clock для синхросигналов проекта.
2. Прописываем констрейны для синхросигналов в соответствии с их реальными значениями частоты.

Не трудно догадаться, что желание разбираться с проектом из-за описанных выше недостатков может быть крайне мало. Приведём в качестве примера ниже одно из возможных описаний констрейна для синхросигнала частотой 50 МГц:

```
create_clock -name {clk} -period 20.000 -
waveform {0.000 10.000} [get_ports {clk}]
```

Пересинхронизация внешнего сигнала сброса в ПЛИС

Чаще всего внешний сигнал сброса приходит в ПЛИС с кнопки. Такой сигнал для ПЛИС является асинхронным и требует пересинхронизации, например, через два триггера. Если сигнал сброса не пересинхронизировать, то это может привести к метастабильности на выходах триггеров, если фронт сброса из активного в неактивный уровень будет близок к фронту синхросигнала. В качестве примера ниже приводится код с описанием двухтриггерного синхронизатора сброса:

```
module rst_sync (

    reset_in,
    reset_out,
    clk
);

  input   reset_in;   // input async reset
  output  reset_out;  // output sync reset
  input   clk;        // input clock

    //=== Wire's, reg's and etc... ===
    reg [1:0]   sync_reg;

    //=== Assignments ===
    assign reset_out = sync_reg[1];

    //=== Description of logic ===
    always @ (posedge clk)
    begin
        sync_reg <= {sync_reg[0], reset_in};
    end
endmodule
```

В целом даже без пересинхронизации сигнала сброса проект в ПЛИС будет работать и читатели, вероятно, наблюдали это, но по указанной выше причине рано или поздно может произойти сбой, с которым придётся разбираться, если не знать одну из потенциальных первопричин. Подробнее о способах пересинхронизации сигнала сброса можно прочитать в соответствующей литературе, например, в главе 10 книги Advanced FPGA Design Architecture, Implementation, and Optimization от Стива Килтса (Steve Kilts).

Проектирование без симуляции RTL

Симуляция - один из пунктов маршрута проектирования, который можно увидеть, например, в документации от Altera (Intel) (рис. 1)

Однако, иногда этим пунктом пренебрегают и проводят тестирование скомпилированного проекта непосредственно внутренними средствами отладки ПЛИС и/или дополнительным оборудованием, что как минимум усложняет отладку и увеличивают её время. Симуляция необходима для проверки работы описанной логики на ранних этапах проектирования. С одной стороны, если логика примитивна, то симуляцией можно пренебречь, но так ли часто она бывает примитивна? Тем более в средах разработки есть возможность генерации шаблонов тестбенчей, упрощающая подготовку к симуляции. Поэтому, по скромному мнению автора, ссылаясь на разные источники, симуляцией пренебрегать не стоит.

Неиспользование внутренних средств отладки

Иногда можно столкнуться с ситуацией, когда ПЛИС для которого делается проект, позволяет подключать внутренние средства отладки. Например, логический анализатор SignalTapII или Source and probes, если говорить про ПЛИС от Intel. При этом у разработчика даже могут быть свободны ресурсы в ПЛИС для внедрения данных средств в проект. Однако, разработчик не изучает как их использовать, не пользуется ими, даже если нет весомых причин ими не пользоваться. Заместо этого отдаётся предпочтение внешним альтернативам, например, осциллографу. В принципе можно сказать, что разработчик удовлетворён данной ситуацией, следовательно, ему нет необходимости что-либо менять в подходах, если это давний, налаженный процесс, а сроки выполнения задачи поджимают. Если взглянуть с другой стороны, то упускается возможность получать информацию о работе внутренних блоков, например: состояния сигналов, регистров и прочего без

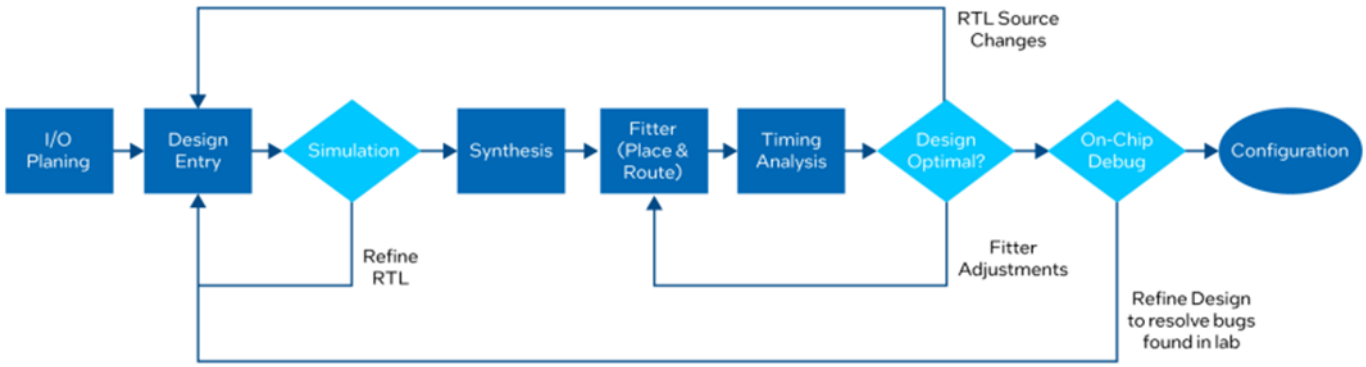


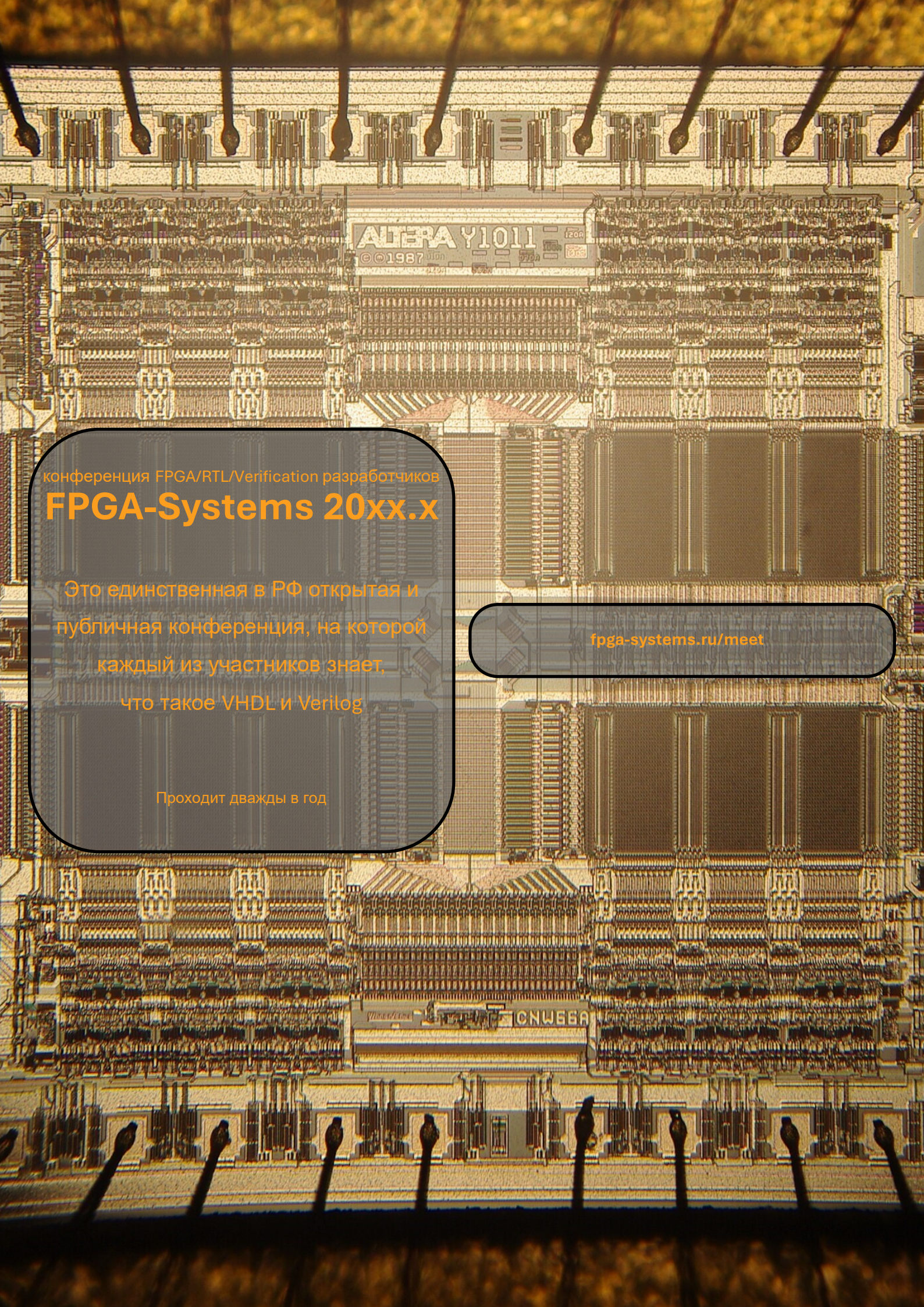
Рисунок 1.

задействования на это портов ПЛИС и дополнительного оборудования. Для этого понадобится лишь штатный JTAG-программатор.

После погружения в мир ПЛИСов, после первых пробных проектов, когда они становятся достаточно крупными, состоящими из объединённых между собой модулей, по скромному мнению автора стоит обратить своё внимание на имеющиеся возможности внутренней отладки проекта со следующими вопросами:

- Какие средства внутренней отладки доступны в ПЛИС, используемой в проекте?
- Какие сигналы, регистры и пр. необходимо подключить в средства отладки?
- Сколько ресурсов ПЛИС они потребуют? Хватит ли ресурсов для описываемой логики и вместе с ней для средства отладки?
- Сможет ли скомпилироваться проект с подключёнными средствами отладки? Нужно ли будет вносить существенные правки в логику?
- Какое средство будет наиболее оптимальным?

Проработка данных вопросов поможет заранее продумать разработчику какие ключевые сигналы, регистры и пр. ему необходимо отследить, отладить в проекте и оценить необходимое для этого средство отладки.



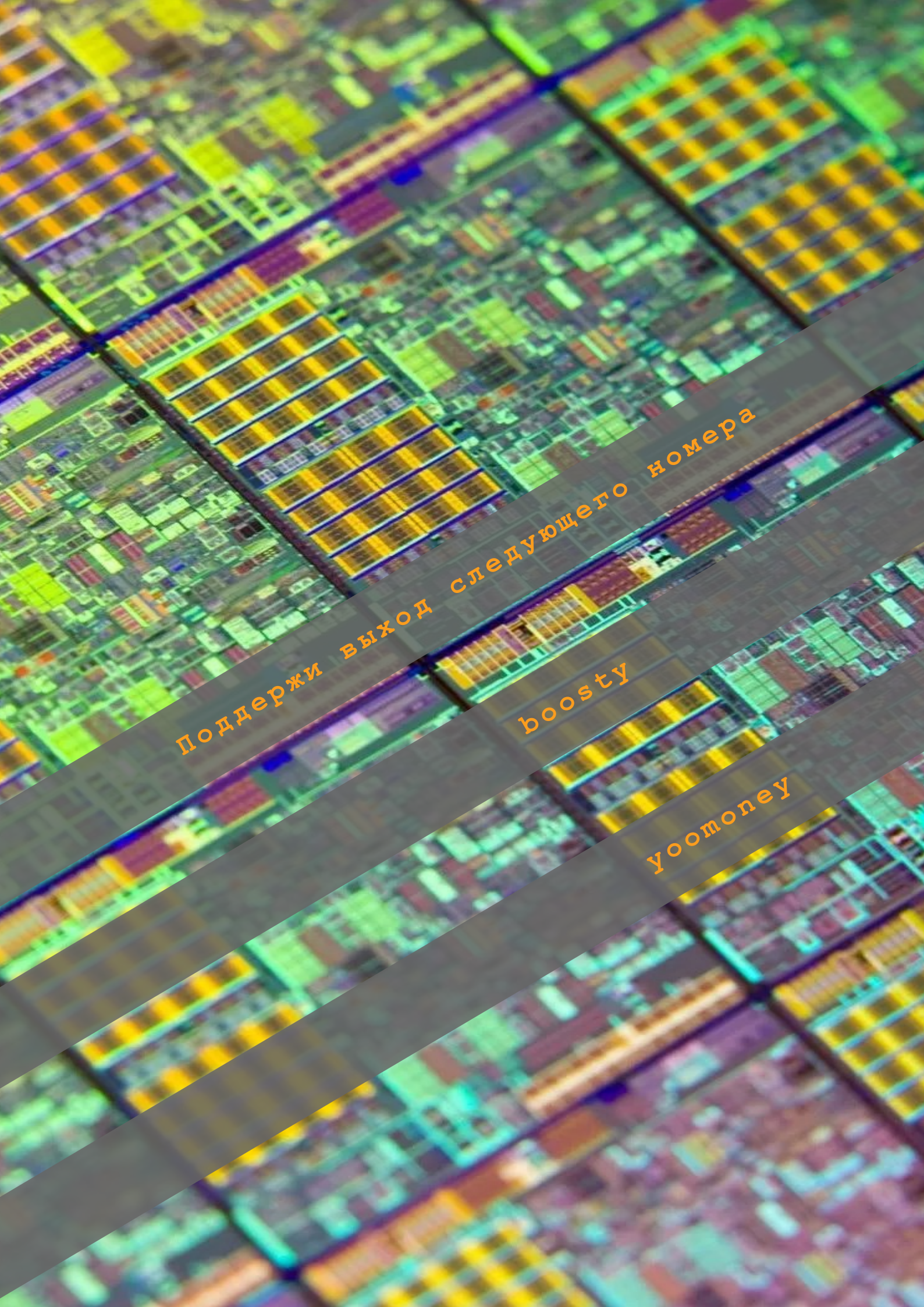
конференция FPGA/RTL/Verification разработчиков

FPGA-Systems 20xx.x

Это единственная в РФ открытая и публичная конференция, на которой каждый из участников знает, что такое VHDL и Verilog

Проходит дважды в год

fpga-systems.ru/meet


Поддержи выход следующего номера
boosty
yoomoney

Профильные телеграм каналы и чаты

@fpgasystems	основной телеграм-чат, в котором задаются вопросы по разработке на ПЛИС.
@fpgasystems_embd	телеграм-чат, где обсуждается разработка для процессорной части систем на кристалле (таких как Zynq-7000, Zynq MP, Intel SoC и др.) или софт-процессоров (например, Nios II, Microblaze, RISC-V, MIPS и др.). Также здесь можно получить ответы на вопросы, связанные с C, Linux и т. д.
@fpgasystems_verification	телеграм-чат по верификации. Здесь обсуждают вопросы, связанные с тестовыми покрытиями, интеграцией Verification IP, использованием верификационных библиотек и фреймворков (UVM, OVM, UVVM, OSVVM, AVM, eRM, URM, RVM, RMM, VMM и др.).
@fpgasystems_dsp	телеграм-чат, где обсуждаются вопросы по цифровой обработке сигналов и их последующей реализации на ПЛИС.
@fpgasystems_events	информационный канал, в котором публикуются свежие новости, анонсы мероприятий и вебинаров, выход новых видео, статей и книг, вакансии и т.д. по проблематике проектирования FPGA/RTL/Верификации
@cpu_design	телеграм-канал, который ведет амбассадор RISC-V International и рассказывает о магии процессоростроения;
@zynq7000	телеграм-канал, посвященный разработкам на Zynq-7000. Автор также рассказывает о работе и с другими платформами
@verif_for_all	телеграм канал одного из преподавателей «Школы синтеза цифровых схем», где просто и понятно рассказывается о верификации цифровых устройств;
@positiveslack	телеграм канал по проблематике ASIC, FPGA, SystemVerilog, UVM; здесь обсуждают цифровой дизайн с уклоном в верификацию
@embedoka	авторский телеграм-канал embedded-инженера
@vlsihub	авторский телеграм-канал о чипмейкерстве и ПЛИСоводстве
@enginegger	телеграм-канал, ориентированный на использование открытых инструментов для FPGA/RTL-разработок и верификации
@docstech_offical	Сайт-документация. Перевод ОФИЦИАЛЬНЫХ документов, руководств и гайдов на русский язык. https://docstech.ru

КОНТАКТЫ | CONTACTS

admin@fpga-systems.ru

Почта издателя | Publisher's e-mail

@fpgasystems_fsm

Канал обсуждения статей из журнала в телеграм
Telegram discussion group

FPGA-Systems.ru/fsm

Сайт журнала | Magazine web-page

@fpgasystems

Телеграм FPGA комьюнити | Telegram of FPGA community

Тел | Phone :: +7-929-955-68-75

FPGA-Systems Magazine :: FSM :: № GAMMA (state_2)
Первый журнал о программируемой логике