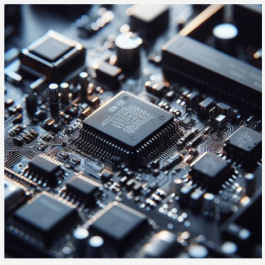
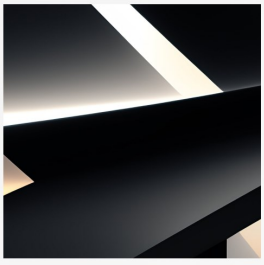
STATE_0

FPGA SYSTEMS

FPGA-Systems
magazine

FSM :: № ALFA





Первый журнал о программируемой логике

ПЛИС-культ привет FPGA комьюнити!

На своих экранах вы листаете первый номер журнала, полностью посвященного программируемой логике.

Путь к его реализации был долог и тернист и в общей сложности занял около трёх лет безуспешных попыток сплотить под одним пером FPGA разработчиков. Но эти попытки оказались не безуспешными, с чем всех нас я и поздравляю.

Workflow нашего электронного издания:

1. журнал разбит на тематические разделы, которые формировались по мере поступления материалов;
2. у каждой статьи есть ссылка для комментариев и обсуждения. Это еще одно большое отличие от печатного издания: мы сделали отдельный канал в телеграм и открыли доступ к обсуждению. Ссылка на обсуждение находится справа под заголовком статьи;
3. у статей в журнале могут быть правки и дополнения поэтому перед прочтением убедитесь, что перед вами актуальная версия номера, которая доступна для скачивания по ссылке fpga-systems.ru/fsm ;
4. читатели могут не только вступить в дискуссию с авторами, но также и поддержать их материально. Ссылки для поддержки авторов прикреплены в постах обсуждения статей.

В журнале есть несколько рекламных баннеров — это небольшая благодарность компаниям, которые всячески помогали нам с организацией мероприятий на протяжении 6 лет существования комьюнити FPGA—Systems. Буду крайне признателен, если вы пройдете по оставленным ссылкам и ознакомитесь с предложениями наших друзей.

Лично от себя: я не уверен, что будет второй номер, раскачать комьюнити на создание полезных заметок не такая простая задача, как казалось в начале. Но если “тайминги сойдутся” и читатели проникнуться идеей выхода периодического FPGA журнала, то я (да и не только я, все мы) будем рады.

ОГРОМНОЕ СПАСИБО всем авторам, которые нашли время и силы на разработку статей для первого номера журнала полностью посвященного программируемой логике FPGA-Systems Magazine :: № ALFA (state_0).

Отдельная благодарность мэтрам FPGA движения, на книгах которых многие из нас выросли, не оставшихся в стороне и решивших поддержать контентом выход первого номера журнала. Пожелаем им здоровья и дальнейших творческих успехов.

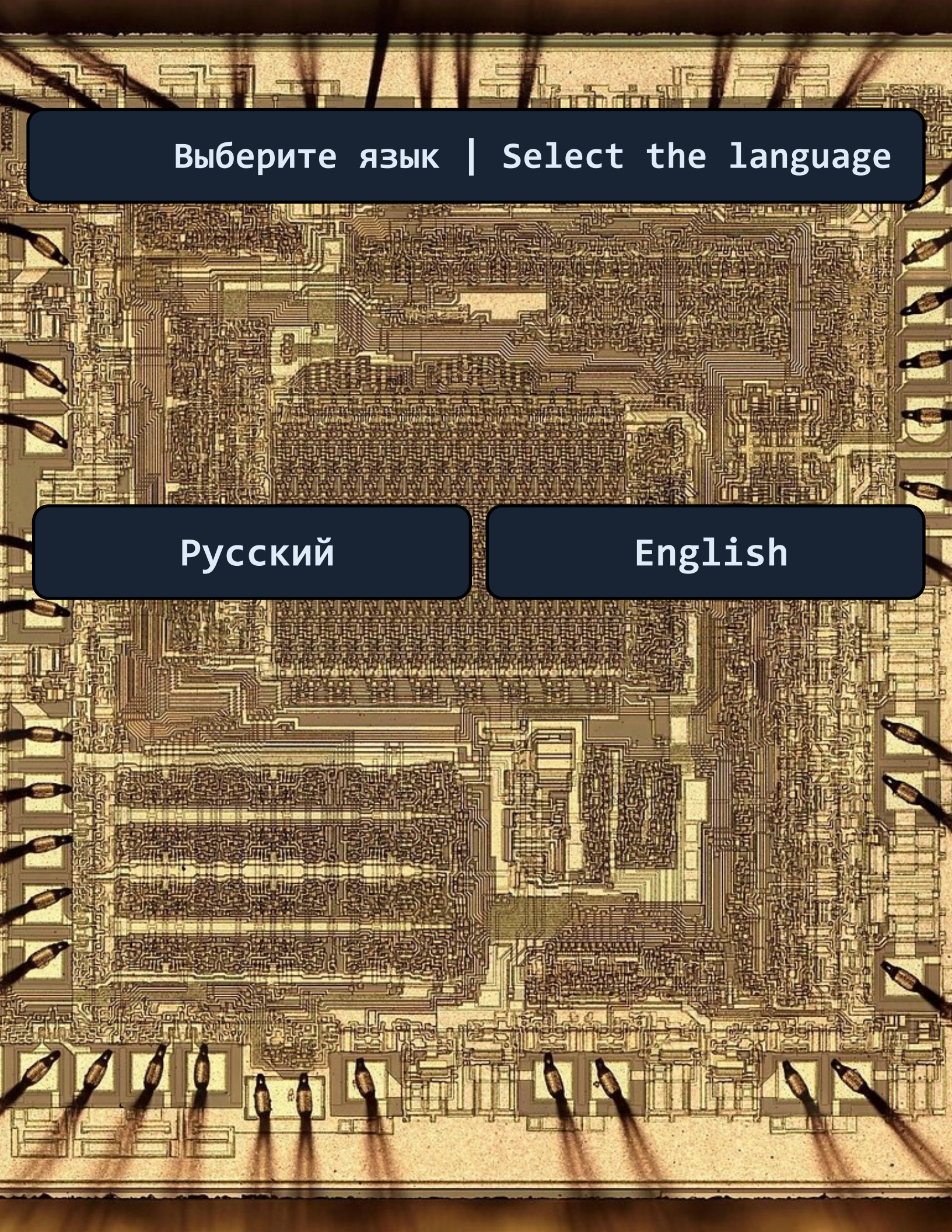
За сим разрешите больше вас не задерживать и пожелать приятного прочтения. Надеюсь, что на страницах журнала читатель найдёт для себя много полезного и нового.

===

С уважением, вождь FPGA комьюнити
Коробков Михаил

Контакты

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


Выберите язык | Select the language
Русский
English

СОДЕРЖАНИЕ НОМЕРА

АНАЛИТИКА | НАЧИНАЮЩИМ | ТУТОРИАЛ | ИССЛЕДОВАНИЯ | РЕАЛИЗАЦИЯ | TIPS & TRICKS

Панчул Юрий.
Что умеют и не умеют писать на SystemVerilog для ASIC и FPGA американские студенты?
click

Хлуденьков Александр
Реализация нейронных сетей на FPGA
click

АНАЛИТИКА | НАЧИНАЮЩИМ | ТУТОРИАЛ | ИССЛЕДОВАНИЯ | РЕАЛИЗАЦИЯ | TIPS & TRICKS

Солодовников А.П.
FPGA 101
click

Балакший Сергей
Зажигаем светодиод процессором J1
click

АНАЛИТИКА | НАЧИНАЮЩИМ | ТУТОРИАЛ | ИССЛЕДОВАНИЯ | РЕАЛИЗАЦИЯ | TIPS & TRICKS

Куренков Константин
Работа с DPI.
click

Аверченко А.П.
Простое вхождение в цифровую схемотехнику с DEEDS
click

АНАЛИТИКА | НАЧИНАЮЩИМ | ТУТОРИАЛ | ИССЛЕДОВАНИЯ | РЕАЛИЗАЦИЯ | TIPS & TRICKS

Бибило П.Н.
Минимизация алгебраических представлений систем булевых функций при синтезе схем модулярных сумматоров и умножителей
click

Мальчуков А.Н.
Разница восприятия САПР QUARTUS языков SystemVerilog и VHDL и разница между VIVADO
click

Соловьев В.В.
Стили и способы описания конечных автоматов на языках Verilog и SystemVerilog
click

Алексеев К.Н., Сорокин Д.А.
Оптимизация вычислительных структур под архитектуру ПЛИС XILINX
click

АНАЛИТИКА | НАЧИНАЮЩИМ | ТУТОРИАЛ | ИССЛЕДОВАНИЯ | РЕАЛИЗАЦИЯ | TIPS & TRICKS

Попов М.А., Романов А.Ю.
Реализация видеовывода сверхвысокой четкости на микросхемах семейства Zynq-7000
click

Сухачев К.И. Шестаков Д.А.
Многоканальное устройство записи (МУЗА_4К10М1)
click

Афанасьев Никита
Реализация интерполятора на платформе SDR Pluto+
click

Минаев Александр
Реализация передатчика MIPI CSI-2 на GOWIN GW2A с подключением к Raspberry PI
click

Гуров В.В.
PYNQ для систем-на-кристалле на примере реализации множества Мандельброта
click

Бортников А.Ю.
Реализация Avalon-MM Master в виде конечного автомата на VHDL.
click

Коробков М.А.
Умножай эффективно. Алгоритм Карацубы. Прямая реализация.
click

Кашпурович В.В.
Интеграция гигабитного последовательного интерфейса на основе стандарта JESD204B: расширение горизонтов передачи данных в ПЛИС
click

Борисенко Н.В.
Мост сопряжения внутрикристального системного интерфейса AMBA APB4 с интерфейсом стыка простого исполнителя STI 1.0
click

Мыцко Е.А.
Аппаратная реализация на ПЛИС свёрточных нейронных сетей для семантической сегментации снимков леса
click

АНАЛИТИКА | НАЧИНАЮЩИМ | ТУТОРИАЛ | ИССЛЕДОВАНИЯ | РЕАЛИЗАЦИЯ | TIPS & TRICKS

Пузанов Николай
Об использовании фильтров в GTKWave
click

Кашканов Артём
Verilator – многофункциональный инструмент эмуляции и тестирования Verilog-кода.
click

Коробков Михаил.
set set set; #это не только легально, но и полезно
click

Высоцкий Матвей
Язык SystemRDL в разработке ip-блоков
click

Что умеют и не умеют писать на SystemVerilog для ASIC и FPGA американские студенты?

Юрий Панчул
Фронт-энд разработчик блоков ASIC
и соавтор Школы Синтеза Цифровых Схем

Телеграм: @yuri_panchul

Обсуждение и комментарии :: ссылка

Введение

За 30 лет работы в Silicon Valley мне пришлось проинтервьировать на позиции в R&D порядка сотни инженеров, половина из которых была недавними студентами. Сначала я работал в области автоматизации проектирования (EDA - Electronic Design Automation), затем в верификации (SystemVerilog, UVM, Bus Functional Models, CPU verification), затем в проектировании (RTL - Register Transfer Level, микроархитектура CPU, GPU, блоков сетевых чипов). Соотвественно я и интервьировал в этих трех областях и накопил некоторую статистику, своего рода взгляд слепого мудреца на слона из индийской притчи.

На этом месте (а может даже сразу по прочтению заголовка) некоторые читатели решат “о, сейчас будут ругать американскую систему образования и превозносить советскую”, после чего примутся, не читая дальше, строчить аргументы в защиту Америки “посмотрите какие они делают айфоны и теслы, а в СССР делали только микроши и агаты”. Боже упаси! Я считаю американские университеты лучшими на планете, а их студентов - сливками, собранными со всех континентов (многие из выпускников Electrical Engineering и Computer Science до учебы в США получили степень бакалавра в топ-университетах Китая, Индии и Европы).

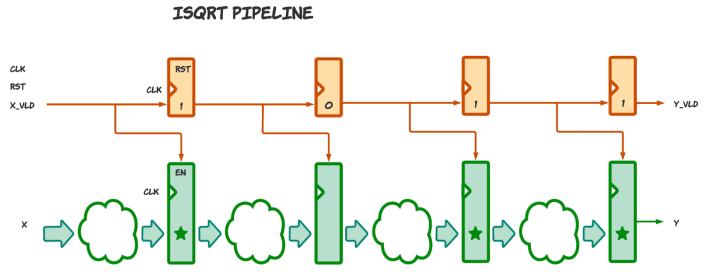
Так чего же они не умеют? Если кратко - многие не умеют решать микроархитектурные задачки с конвейерами и очередями FIFO, а также верифицировать свой дизайн тестбенчами с самопроверкой. Среди курсовых проектов в их резюме может стоять реализация блока суперскалярного процессора для внеочередных вычислений (алгоритм Томасуло), но при этом студенты могут ставится в тупик предложением построить конвейерный блок (нарисовать микроархитектурную диаграмму и написать код на верилоге) для вычисления какой-нибудь плоской формулы типа sqrt (a + sqrt (b)), где sqrt - это конвейерный подблок с фиксированной латентностью, берущий квадратный корень от числа.

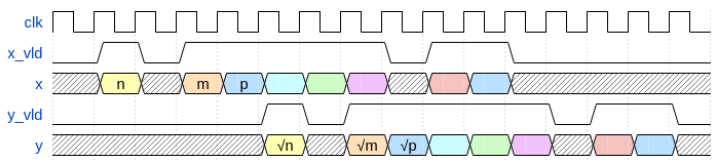
Такое неумение разумеется создает проблему для любого работодателя - будь то крупная электронная компания или стартап в области аппаратного ускорителя ML. Потому что получается, что студенты 4 или 6 лет учили научпоп, потратили на него несколько сотен тысяч долларов денег своих родителей или влезли в долги сами - а теперь электронные компании должны их учить что-то делать руками с нуля, в условиях жестких дедлайнов массовых продуктов.

Если построить образовательную систему, в которой образование будет происходить в образовательных учреждениях, а не после окончания образования, то можно существенно повысить конкурентоспособность сопряженной промышленности. Но перейдем к конкретике.

Основа для задач этой заметки - блок ISQRT

Допустим у вас есть конвейерный блок ISQRT с фиксированной латентностью N, который вычисляет целочисленный квадратный корень. “Фиксированная латентность N означает”, что после того, как в блок входит число X, на его выходе Y через N тактов сигнала CLK появляется число Y=√X. На число N мы будет также ссылаться далее как на “глубину конвейера” или “количество стадий конвейера”.





Вместе с данными X и Y, каждое шириной несколько бит, используются два однобитовых контрольных сигнала - X_VLD и Y_VLD. Сигнал *_VLD (valid) означает, что соотвествующее данное “живое”, а не является просто находящимся на шине мусором. Хотя блок может работать и без таких сигналов (напомним что его латентность N фиксирована), но наличие сигналов valid полезно по двум причинам:

1. Y_VLD дает понять другому блоку, который использует число Y, выходящее из ISQRT, что это данное Y готово. Конечно другое блок может в принципе сам определить это, используя момент входа и знание, что Y выйдет через N тактов после вхождения X. Но разработчик второго блока может не хотеть полагаться на такое допущение - во-первых, расчитывая заменить блок ISQRT на блок с переменной латентностью, а во-вторых с Y_VLD его логика может быть проще.

2. Наличие сигнала X_VLD позволяет экономить динамическое энергопотребление. Если X_VLD стоит в 0, то данные X не записываются в регистр (группу D-триггеров) первой стадии конвейера. X_VLD подключается к сигналу EN (enable) D-триггеров данных. Когда EN стоит в 0, то D-триггер не реагирует на тактовый сигнал CLK. Эта же идея применяется для всех стадий, на которых тоже есть сигналы valid, все вместе объединенные в сдвиговый регистр.

   Таким образом экономятся пиковатты на переключение D-триггеров данных. Затем из всех D-триггеров большого дизайна на миллиарды транзисторов экономия становится уже значимой: удлиняется время жизни на батарейке для мобильных чипов и отменяется необходимость делать жидкостное охлаждение для чипов сетевых роутеров и ускорителей ML в датацентрах.

Также заметим, что для D-триггеров, в которые записываются контрольные сигналы valid, необходим сигнал сброса RST, а для D-тригерров, в которых хранят промежуточные данные - нет. Потому что находится ли в них живые данные или мусор - определяется сигналом valid.

Интересно, что американские студенты даже с такими простыми дизайнами допускают три шероховатости. Это не фатально, но шероховатости почему-то кочуют из методички в методичку лет 25 - профессорам наверное лень это исправить.

1. Помещают присваивание регистрам для valid и регистрам для данных в одном always блоке. Почему это плохо, по крайней мере для синхронного сброса, разобрано в статье Клиффа Каммингса: Cliff Cummings. “Synchronous Resets? Asynchronous Resets? I am so confused! How will I ever know which to use?”, глава “2.1 Synchronous reset flip-flops with non-reset follower flip-flops”.

2. Ставят ненужный сброс для регистров, в которых хранят данные, при том, что параллельно с этими регистрами есть контрольные сигналы valid. Однобитовый valid при сбросе устанавливается в 0 и тем самым указывает, что значение сброшенных данных нужно игнорировать (см. объяснение выше).

3. Используют сброс активный при значении 0, а не 1 (active low, а не active high). То есть везде пишут “if (! reset)”, а не “if (reset)”. При том, что в индустриальном коде процессорных и сетевых компаний весь 21 век в бОльшей части кода пишут сброс с положительной полярностью, а на отрицательную меняют только на внешнем уровне системы на кристалле, когда приходит сигнал из внешнего по отношению к чипу мира.

   Этот active low reset пошел от электрических соображений для внешних сигналов, но протаскивать его через миллионы строк кода чипа для телефона или роутера совершенно бессмысленно. Даже если где-то для внутренних стандартных ячеек ASIC (ASIC standard cell) будет предпочтителен active low reset, то программа синтеза поменяет полярность сброса автоматически.

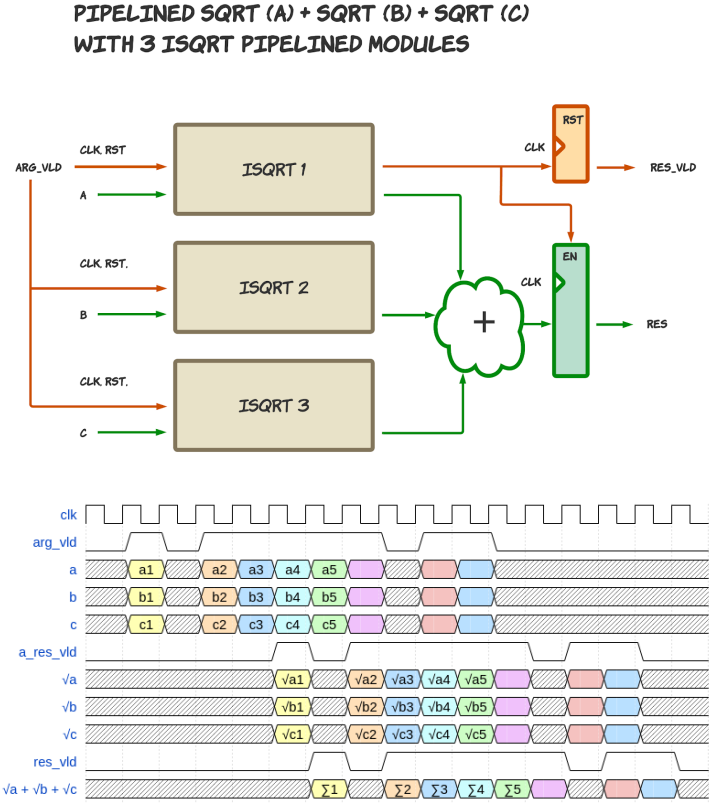
Все что я написал до сих пор - это предисловие, задачи сейчас будут. Сам блок ISQRT, который используется в задачах, я написал на основе алгоритма, описанного в книжке Hacker's Delight by Henry Warren. Хотя это книжка для программистов, но в ней есть и примеры, полезные для электронщиков.

Эта реализация ISQRT находится в GitHub репозитории для домашних заданий Школы Синтеза Цифровых Схем. В некоторых реализациях подобных блоков предпочитают ставить регистры перед комбинационной логикой, то есть сразу класть в регистры приходящие аргументы. Для целей нашей заметки это несущественно - можно делать и так, и так. Синтез все равно сделает всю модульную иерархию дизайна плоской и будет вычислять задержки между D-триггерами глобально.

Вопрос на интервью 1: Конвейер для √A + √B + √C

Формулировка: Допустим, у вас имеется конвейерный блок для вычисления квадратного корня с фиксированной латентностью N. Спроектируйте конвейерный блок для вычисления значения выражения √A + √B + √C, где аргументы A, B и C приходят одновременно, и в каждом такте появляется новая тройка этих аргументов. Какая будет у этого конвейера латентность?

Боьшинство студентов, которые изучали верилог, решают эту задачу без проблем. Они создают три экземпляра блока ISQRT, подают им на вход valid сигнал для аргументов, суммируют результаты, выходящие из трех блоков и кладут результат в регистр. После чего говорят что латентность получившегося блока N + 1.



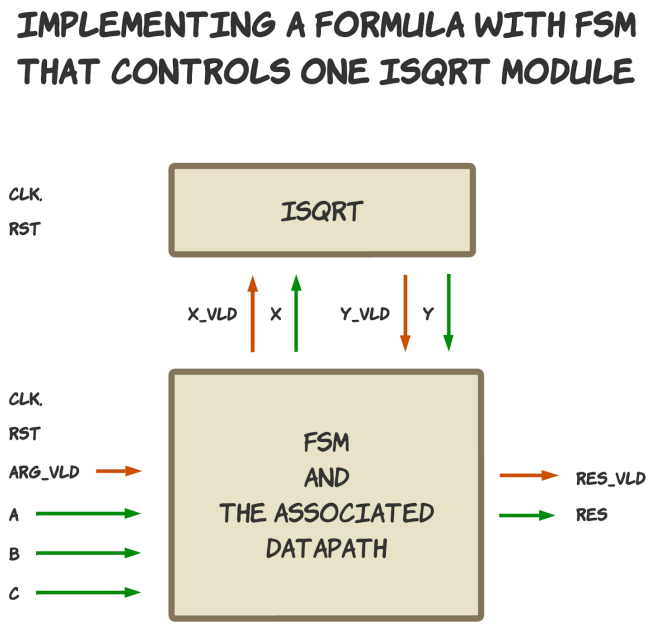
Вопрос на интервью 2: Конечный автомат для вычисления √A + √B + √C, когда можно использовать только один экземпляр модуля квадратного корня

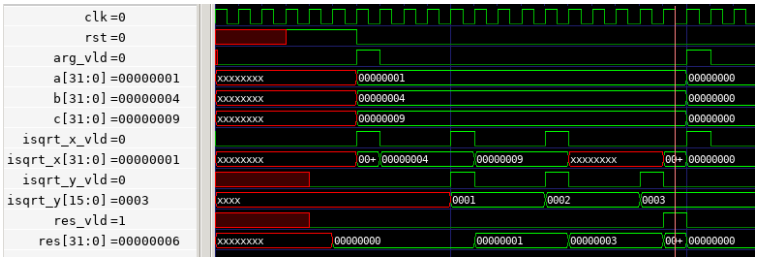
Формулировка: Теперь предположим, вам нужно вычислить сумму трех квадратных корней, но вы можете использовать только один экземпляр модуля ISQRT. Можно ли это сделать и если да, то какая будет латентность у спроектированного блока?

Как правило, студенты предлагают реализовать такое решение с помощью конечного автомата (Finite State Machine - FSM), который:

1. запускает вычисление для аргумента A;
2. ждет получения результата √A;
3. сохраняет √A и запускает вычисление для аргумента B;
4. ждет получения результата √B;
5. суммирует √B с ранее сохраненным √A, сохраняет сумму и одновременно запускает вычисление для аргумента C;
6. ждет получения результата √C;
7. и наконец суммирует √C с сохраненным √A + √B, записывает в регистр RES результат и выставляет бит RES_VLD.

После чего говорят, что латентность блока 3*N+1.

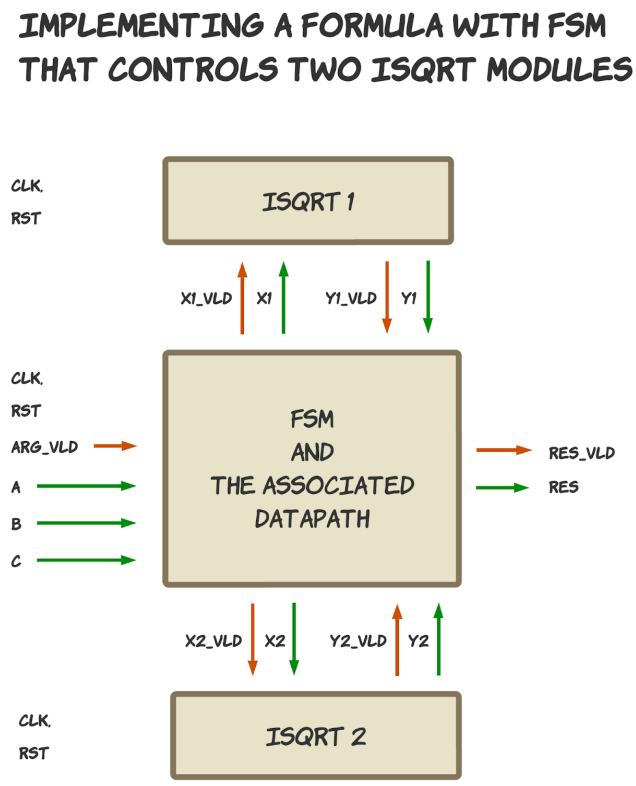




В таком решении есть неоптимальность (заметили?), но прежде чем указать на нее студенту, я задаю дополнительный воспрос:

Вопрос на интервью 3: Конечный автомат для вычисления √A + √B + √C, когда можно использовать два экземпляра модуля квадратного корня

А что если модуля ISQRT два? С помощью изменения конечного автомата студент делает решение, при котором вычисления корней для A и B происходят одновременно, а вычисление √C начинается, после получения результатов √A и √B. На этом студент рапортует латентность 2*N+1:



Вопрос на интервью 4: Можно ли оптимизировать ваш конечный автомат с учетом факта, что ISQRT - конвейерный модуль?

Проблема с ответами 3*N+1 и 2*N+1 заключается в том, что студент забыл, что модуль ISQRT - конвейерный, и следовательно, вычисления для √B и √C можно запустить, не дожидаясь окончания вычисления √A. То есть можно изменить конечный автомат для работы с одним блоком ISQRT следующим образом:

1. запускаем вычисление для аргумента A;
2. в следующем такте запускаем вычисление для аргумента B;
3. еще в следующем такте запускаем вычисление для аргумента C;
4. ждем получения результата √A и записываем его в регистр;
5. в следующем такте после получения результата √A получаем результат √B, складываем с ранее записанным √A и записываем сумму в регистр;
6. еще в следующем такте получаем результат √C, складываем с ранее записанной суммой √A+√B, записываем сумму в регистр RES и выставляем бит RES_VLD.

Латентность такого вычисления будет всего N+3, это сильное улучшение по сравнению с 3*N+1, при практически тех же аппаратных ресурсах. Интересно что добавление второго экземпляра блока ISQRT улучшает нашу латентность всего на такт, она становится N+2. Но цена такого улучшения - целый тяжелый арифметический блок.

Впрочем, это не сильно большая ошибка. Настоящие проблемы у студентов начинаются, когда между вычислениями появляются зависимости.

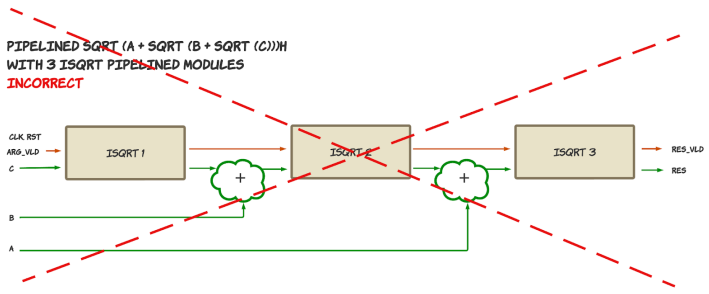
Вопрос на интервью 5: А как вы будете вычислять √(A + √(B + √C))?

“Да так же само”, - говорит студент, - “просто немного поменяю конечный автомат”.

“Ну хорошо”, говорю я, и задаю следущий вопрос:

Вопрос на интервью 6: А можно ли организовать конвейер для вычисления √(A + √(B + √C)) так, чтобы тройка A, B и C приходила каждый такт, и каждый такт появлялся был результат RES?

Некторые студенты начинают говорить “это невозможно”, другие сначала рисуют вот такую диаграмму:



На что я им говорю: “но если вы реализуете такую схему, и подадите на ее вход аргументы A1, B1 и C1, а в

следующем такте A2, B2 и C2, итд, то на вход блока ISQRT 2 через N тактов придет не аргумент B1, соотвествующий аргументу C1, а аргумент B N+1 - и получится неверный результат.

Тут начинается “а можно считать ISQRT комбинационным блоко?”. Отвечаю “нет, нельзя, это было бы слишком просто”.

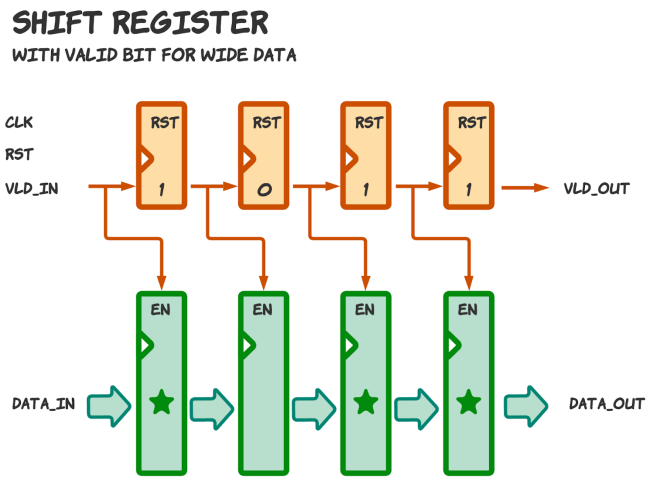
“А может использовать здесь алгоритм Томасуло?” - спрашивает студент. Это метод организации вычислений в суперскалярном процессоре, который обьъясняют студентам на лекциях, и по которому они делают курсовые проекты. Я объясняю, что это слишком тяжелый в смысле ресурсов метод, который нельзя тащить в каждый случай простого статического конвейера. Это как прикрутить карьерный самосвал к велосипеду, чтобы он быстрее ездил.

“А может записать B в промежуточный регистр?” - предполагает студент. Отвечаю, что он мыслит в правильном направлении, но одного регистра недостаточно, так как этот регистр будет переписан через такт третьим данным. “То есть нужно несколько регистров?” - удивляется студент, - “но это какая-то бессмыслица” - качает он голову.

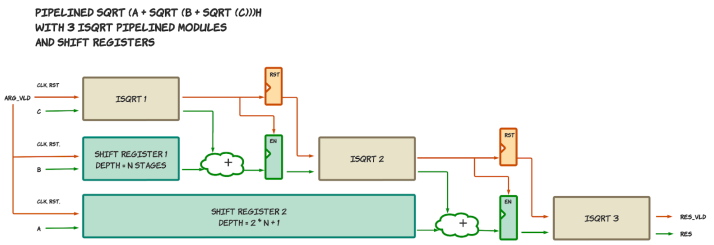
“А преподавали ли у вас в университете сдвиговые регистры или там очереди FIFO?” - спрашиваю я. Ведь и одно, и другое можно использовать, чтобы задержать аргумент B на N тактов.

Тут выясняется, что про сдвиговые регистры (shift register) студенты просто забыли. Видимо это было в начале курса введения в цифровую логику, и они их просто пропустили, как какую-то не привязанную к жизни сущность. Такие сущности существуют - например по непонятной мне причине из курса в курс профессора разбирают JK-триггеры, которые в реальной жизни проектировщика для ASIC или FPGA не появляются. В 99% элементов состояния используются D-триггеры, гораздо реже (для clock gating, latch arrays или time borrowing) D-защелка.

Но сдвиговые регистры, составленные из D-триггеров - это совсем другое дело. Их можно использовать, чтобы тащить данные параллельно конвейеру. Фактически, сдвиговый регистр - это сам конвейер, но без вычислений - данные в нем просто задерживаются на N тактов.



Со сдвиговыми регистрами конвейерный вычислитель формулы √(A + √(B + √C)) строится без проблем:

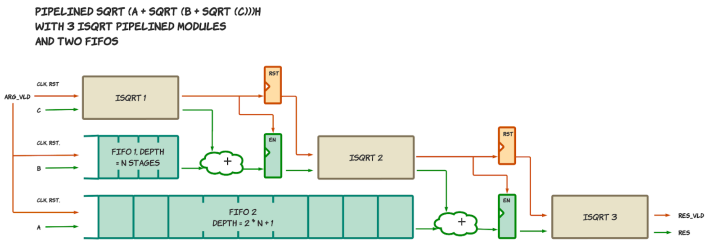


Вопрос на интервью 7: А можно ли использовать для вычисления конвейерного √(A + √(B + √C)) - не сдвиговые регистры, а FIFO?

Сдвиговые регистры (которые в контексте конвейеров часто называют staging registers) - это не единственный способ решения проблемы потактового выравнивания аргументов. Для широких данных и глубоких конвейеров, а также если латентность конвейера не фиксирована - предпочтительнее использовать очереди FIFO. Это экономит динамическое энергопотребление, так как в FIFO меняются небольшие указатели, а в сдвиговом регистре - двигаются большие данные .

Тут я часто спрашиваю у студентов, учили ли они в университете FIFO. Оказывается, учили и даже могут сказать, какие у FIFO сигналы. Вот только это FIFO висит у студентов в голове в пустоте - они не знают в каких случаях его применять. А ведь FIFO годится не только чтобы например хранить данные, выходящие из конвейера, но и для нашего случая - хранить часть данных параллельно конвейеру, когда другая часть идет через конвейер. В частности, можно поставить FIFO параллельно модулю ISQRT 1 и вталкивать в него аргумент B, когда мы запускаем соотвествующее C в

конвейер блока ISQRT 1. А потом выталкивать, когда мы получаем результат √C, их складывать и запускать сумму в блок ISQRT 2.



Но и это еще не все.

Вопрос на интервью 8: А можно ли сделать конвейерный вычислитель из неконвейерных модулей?

Предположим, что модуль ISQRT реализован не с помощью конвейера, а конечным автоматом, то есть не может принимать данные каждый такт, а требует ожидания N тактов перед подачей ему нового запроса. Можно ли из таких модулей построить конвейерный вычислитель квадратного корня, который принимает аргумент каждый такт и каждый такт выдает результат?

Это вопрос уже не для студентов, а для старших инженеров. Но ответ на него - да, можно. Нужно просто сделать несколько экземпляров (instances) неконвейерного вычислителя, и каждый так сбрасывать новый запрос на вычисления на незанятый экземпляр, а также отправлять только что вычисленный результат с отработавшего экземпляра.

Это один из случаев так называемой техники упрятывания латентности (latency hiding). Он применяется не только с вычислительными блоками, но и например при работе с памятью.

Вопрос на интервью 9: А что если другой модуль не всегда может принять результат вычисления формулы? Как затормозить работу конвейера, чтобы результаты вычислений не выбрасывались?

Эта проблема называется “давление назад” или “back-pressure”. Я не ожидаю от студентов написания конвейеров с back-pressure во время интервью. Разве что в формате домашнего задания. Но каждый твердый мидл-разработчик должен владеть приемами back-pressure: протоколом valid/ready, кредитными счетчиками итд. Но это тема отдельной заметки.

Заключение

В этой заметке я разобрал только один класс задач из десятков, которые должны быть частью вузовских программ по проектированию цифровых схем. Хотел сказать “как взятие интегралов в курсе матана”, но осознал, что это плохое сравнение. 99% студентов, которые учили матан, после университета не берут никаких интегралов никогда. А вот проектировщики процессоров, GPU, сетевых чипов, блоков DSP/ЦОС и ускорителей ML - решают задачи по проектированию конвейеров каждый день и собственно за это и получают зарплату.

Причем понимать это должны не только front-end RTL дизайнеры, но и верификаторы, чтобы представлять как по конвейеру идут транзакции. И даже инженеры которые отлаживают чипы уже изготовленные на фабрике - с этих чипов можно снять через scan chain биты, отслеживающие состояние стадий конвейера, очередей и кредитных счетчиков. Также про конвейеры должны знать программисты которые пишут САПРы для синтеза, симуляции и формальной верификации - иначе у них неправильные приоритеты из-за незнания, какие задачи решает их пользователь - проектировщик микросхем.

Ссылки:

1. PNG файлы с диаграммами https://www.dropbox.com/scl/fo/b5imn8aug66p0npvzm802/h?rlkey=50mo3ec1e7nnziizgvcdbjpbl&dl=0
2. Ссылка на директорию в Lucidspark https://lucid.app/folder/invitations/accept/inv_6bf39b3c-7afa-45fe-817b-f6fd15cc9f85
3. Диаграммы в репозитории Школы Синтеза Цифровых Схем https://github.com/chipdesignschool/systemverilog-homework/tree/main/doc/isqrt
4. Задачи про конечные автоматы в репозитории Школы Синтеза Цифровых Схем https://github.com/chipdesignschool/systemverilog-homework/tree/main/03_finite_state_machines/03_04_sqrt_formula_fsms
5. Cliff Cummings. “Synchronous Resets? Asynchronous Resets? I am so confused! How will I ever know which to use?” https://www.sunburst-design.com/papers/CummingsSNUG2002SJ_Resets.pdf
6. Henry Warren. Hacker's Delight. Addison-Wesley Professional. 2nd edition, September 25, 2012.

школа синтеза цифровых схем

ПРИ ПАРТНЕРСТВЕ

YADRO

Проект, который позволяет специалистам любого уровня — от школьников до практикующих инженеров — получить не только теоретические знания, но и практические навыки в области разработки цифровых микросхем.

Что нужно знать о школе

- Программа рассчитана на 6 месяцев, включает в себя элементы курсов компьютерной архитектуры и микроархитектуры процессорных ядер.
- Лекции ведут опытные специалисты, а часть заданий основаны на задачах ведущих отечественных и мировых микроэлектронных компаний. Школа может стать отправной точкой для карьеры или новой ступенью развития.
- Можно посещать онлайн-трансляции или очные занятия в одном из 22 кластеров в России и Беларуси. География кластеров — от Томска и Новосибирска до Севастополя и Ростова-на-Дону, вся карта — по ссылке.
- Участие бесплатное.

Зарегистрироваться и стать участником Школы синтеза можно в любое время.

Занятия проходят каждую субботу с 12:00 до 15:00 по московскому времени.

ЗАРЕГИСТРИРОВАТЬСЯ

Смотреть записи лекций на YouTube

Реализация нейронных сетей на FPGA

Хлуденьков Александр, *ООО «Лассард».*

e-mail: model3d@mail.ru

Обсуждение и комментарии :: ссылка

В данной обзорной статье автор хотел бы обобщить те сложности и находки, с которыми ему пришлось столкнуться во время обучения и работы. Возможно, данный опыт будет кому-то полезен.

Тема работы нейронных сетей на FPGA широко известна, так как обладает рядом неоспоримых преимуществ. Это практически «аппаратная» реализация, не «псевдо», а самое настоящее распараллеливание процесса вычислений, разработка под каждую архитектуру нейронной сети.

При вводе запроса в интернет-поисковиках «Нейронные сети на FPGA» результат будет весьма обширным, однако автору данной статьи не довелось увидеть реально применяемого примера нейронных сетей, работающих на FPGA.

Хочется отметить, что при разработке нейронных сетей для рабочего использования под каждую конкретную задачу выбирается определенная архитектура (или просто покупается готовое решение) и аппаратное обеспечение, на котором работает сеть. Если это блок обработки видеоинформации узла движения марсохода или блок управления АЭС – то решение, несомненно, за встраиваемыми системами.

При реализации нейронных сетей на системах «классической» архитектуры проявляется большой недостаток – последовательная обработка информации, вследствие чего возникает относительно низкая скорость работы сети, которая возрастает экспоненциально при росте количества нейронов.

Поэтому задача распараллеливания процессов работы сети посредством аппаратной реализации является весьма актуальной, в частности, с применением программируемых логических интегральных схем (FPGA). При обработке информации нейронной сетью производится практически только две простые операции – умножение и суммирование результата (так называемая операция умножения с накоплением). Взятие функции активации занимает малую долю вычислительных ресурсов.

Как известно, в «классических» FPGA нет встроенной арифметики с плавающей точкой. Поэтому следует привести все весовые коэффициенты, которые обычно имеют значение из промежутка (0 … 1), к дискретному значению 0 … n-1, где n – определенная степень числа 2, желательно четная, то есть из ряда 4, 16, 64 и т. п. (для четного числа бит). Произвести так называемое квантование значений весов - умножив число с плавающей точкой на максимально возможное значение веса при целочисленном представлении и затем округлив его в ближайшую сторону. Конечно, в процессе квантования будет происходить потеря точности, но она будет падать при увеличении числа бит на нейрон.

На процесс выбора разрядности нейронов очень сильно влияет разрядность встроенных умножителей (18, 23 и далее бит). На взгляд автора, разрядность весов в 16 бит является оптимальным вариантом, так как в данном случае рассматриваются числа из диапазона (0 … 65535), что является вполне достаточным.

Так же необходимо квантовать функцию активации, которую необходимо будет аппроксимируем в кусочно-линейную непрерывную форму. Этого возможно достичь, используя оператор case, к примеру, таким образом:

```
case (Sum [7:0])
    8'b00000000: begin
        output reg [7:0] = 8'b00000000;
    end
    8'b00000001: begin
        output reg [7:0] = 3'b00000000;
    end
    // перебираются все значения из рабочего
промежутка
    8'b00100100: begin
        output reg [7:0] = 3'b00011101;
    ….
    end
endcase
```

После приведения весов в дискретную форму необходимо перенести архитектуру сети в микросхему FPGA. Возможны два варианта решения данной задачи.

Первый вариант – перенос весов в уже готовой программе, т. е. веса будут зашиты в программу FPGA как константы, прописаны в коде.

Второй вариант – перенос весов нейронов из стороннего источника (по сети, с флеш-накопителя и т. п.). Веса «подгружаются» во время работы программы.

Оценим преимущества и недостатки каждого подхода.

Первый вариант. Весовые коэффициенты «вшиты» в код программы. Код программы будет выглядеть примерно так:

```
inout wire [15:0] W_1_1,  // Объявление веса первой связи
inout wire [15:0] W_1_2, // Объявление веса второй связи
inout wire [15:0] W_1_3, // Объявление веса третьей связи
……
assign W_1_1 = 16'b0111001011000111; // Значение веса первой связи
assign W_1_2 = 16'b1011110001010100; // Значение веса второй связи
assign W_1_3 = 16'b1100111100101011; // Значение веса третьей связи
……..
```

И так далее для всех весовых коэффициентов.

При загрузке в FPGA кода такой структуры, будет реализована наибольшая производительность нейронной сети. При изменении весовых коэффициентов необходимо будет каждый раз загружать данные заново. Но нейронная сеть обычно как раз работает с неизменяемыми весами (не рассматриваем рекуррентные и прочие экзотические виды нейронных сетей).

Второй вариант. На HDL пишется что-то наподобие софт-процессора, который управляет загрузкой весовых коэффициентов со стороннего источника, загружает входной вектор с данными (к примеру, видеокадр), производит умножение и суммирование. Для данного способа замечательно подходят так называемые систолические массивы - однородная сеть тесно связанных блоков обработки данных (DPU), называемых ячейками или узлами. Однако при таком подходе преимущества FPGA практически теряются, так как софт-процессор, пусть даже и специализированный, не в состоянии «угнаться» за своим аппаратным коллегой.

Поэтому единственный путь для разработки реально работающих нейронных сетей на FPGA – это создание программы с занесенными в код весовыми коэффициентами. Каким способом вносить весовые коэффициенты в код программы? Это возможно произвести посредством различных способов.

Во-первых, возможно вручную писать код, занося коэффициенты по очереди из текстового файла. Конечно, это очень долгая и трудоемкая работа и вручную переносить веса практически нет смысла.

Вторым вариантом является тоже ручная разработка на языке HDL, но будут созданы вначале небольшие куски кода - заготовки. Это могут быть фрагменты кода нейрона или целого нейронного слоя.

Часть кода для реализации нейрона на языке Verilog может выглядеть следующим образом.

module NeuronNet (

input reg [7:0] NeuronInput_1, // Значение первого входного сигнала

```
module NeuronNet (
input reg [7:0] NeuronInput_1, // Значение первого входного сигнала
input reg [7:0] NeuronInput_2, // Значение второго входного сигнала
……..…
output reg [7:0] NeuronOutput, // Значение выходного сигнала
…….
inout wire [9:0] Sum); // Значение получившейся суммы
always @* begin
    assign W_4 = 3'b010;
    Sum [9:0] = NeuronInput_1*W1 + NeuronInput_2*W2 + ….
    //  вычисление суммы
    //  Функция активации
end
endmodule
```

Даже если в двух слоях полносвязной сети по 784 нейрона (для работы с «детским» набором MNIST размером 28x28 пикселей), мы получим 614 656 операций умножения. Вручную такой код никто писать не будет!

В настоящее время сложные нейронные сети пишут на высокоуровневых средствах проектирования, таких как Vivado HLS от Xilinx и Simulink (пакет MATLAB). Однако оптимальность полученного кода в таком случае под вопросом.

Намного более лучшим вариантом будет разработка специального кодогенератора на одном из

высокоуровневых языков программирования, к примеру, C#. Данный кодогенератор будет получать на вход архитектуру нейронной сети и весовые коэффициенты. На выходе будет получаться код на одном из языков описания аппаратуры (к примеру, на Verilog).

Стоит отметить, что в настоящее время для серьёзной работы (задачи распознавания и классификации изображений) «классические» полносвязные сети практически никто не применяет. Чаще всего для этого применяют свёрточные нейронные сети, имеющие много слоёв, которые по принципу работы и методам обучения кардинально отличаются от других типов нейронных сетей.

При работе работы нейронной сети с сотней слоев, реализованной на FPGA, время задержки будет уже достаточно большим. Поэтому необходимо использовать конвейер. Блоки синхронизации необходимо ставить после каждого слоя.

Существует еще один не менее важный параметр FPGA – емкость.

Рассмотрим конкретную задачу – распознавание номера въезжающей машины по фото с VGA-видеокамеры. Исходные данные – изображение 640x480, 16 бит. Получившийся входной вектор имеет размер 640 x 480 = 307 200 элементов по 16 бит.

Используем для работы трехслойную сеть. Первый слой – входной, 307 200 нейронов, принимающих и передающих сигнал. Второй слой (скрытый) – установим 1000 нейронов. На выходе получаем вектор вида [цифра, цифра, цифра, буква, буква, буква]. Номер региона для простоты рассматривать не будем. Выходной слой – 120 нейронов.

Подсчитаем количество умножений для передачи сигналов из первого во второй слои. При реализации «классической» полносвязной сети количество связей между первым и вторым слоем слоями нейронной сети составит: 307200 x 1000 = 307 200 000 операций умножения. 300 миллионов! Понятно, что такое количество умножителей является запредельным.

Даже простая реализация полного распараллеливания сети уперлась в ёмкость FPGA. Следовательно, необходимо каждый слой делить на «куски» и производить с ними отдельные вычисления.

Таким образом, на взгляд автора, наилучшим решением для реализации нейронных сетей на FPGA будет комплект из макроблоков, которые реализуют блоки обработки нейронов.

Проработка таких блоков – тема отдельной статьи.

FPGA 101

Солодовников Андрей Павлович

E-mail: hepoh3@gmail.com, Telegram: @HepoH

Обсуждение и комментарии :: ссылка

1. Что такое язык описания аппаратуры (HDL)

На заре появления цифровой электроники, цифровые схемы в виде диаграммы на бумаге были маленькими, а их реализация в виде физической аппаратуры — большой. В процессе развития электроники (и её преобразования в микроэлектронику) цифровые схемы на бумаге становились всё больше, а относительный размер их реализации в виде физических микросхем — всё меньше. На рисунке 1, вы можете увидеть диаграмму цифровой схемы устройства intel 4004, выпущенного в 1971 году.

Данная микросхема состоит из 2300 транзисторов[2].

За прошедшие полсотни лет сложность цифровых схем выросла колоссально. Современные процессоры для настольных компьютеров состоят из десятков миллиардов транзисторов. Диаграмма выше при печати в оригинальном размере займет прямоугольник размером 115x140 см с площадью около 1.6м^2. Предполагая, что площадь печати имеет прямопропорциональную зависимость от количества транзисторов, получим что печать диаграммы современных процессоров потребует площадь в 16млн. м^2, что эквивалентно квадрату со стороной в 4км.

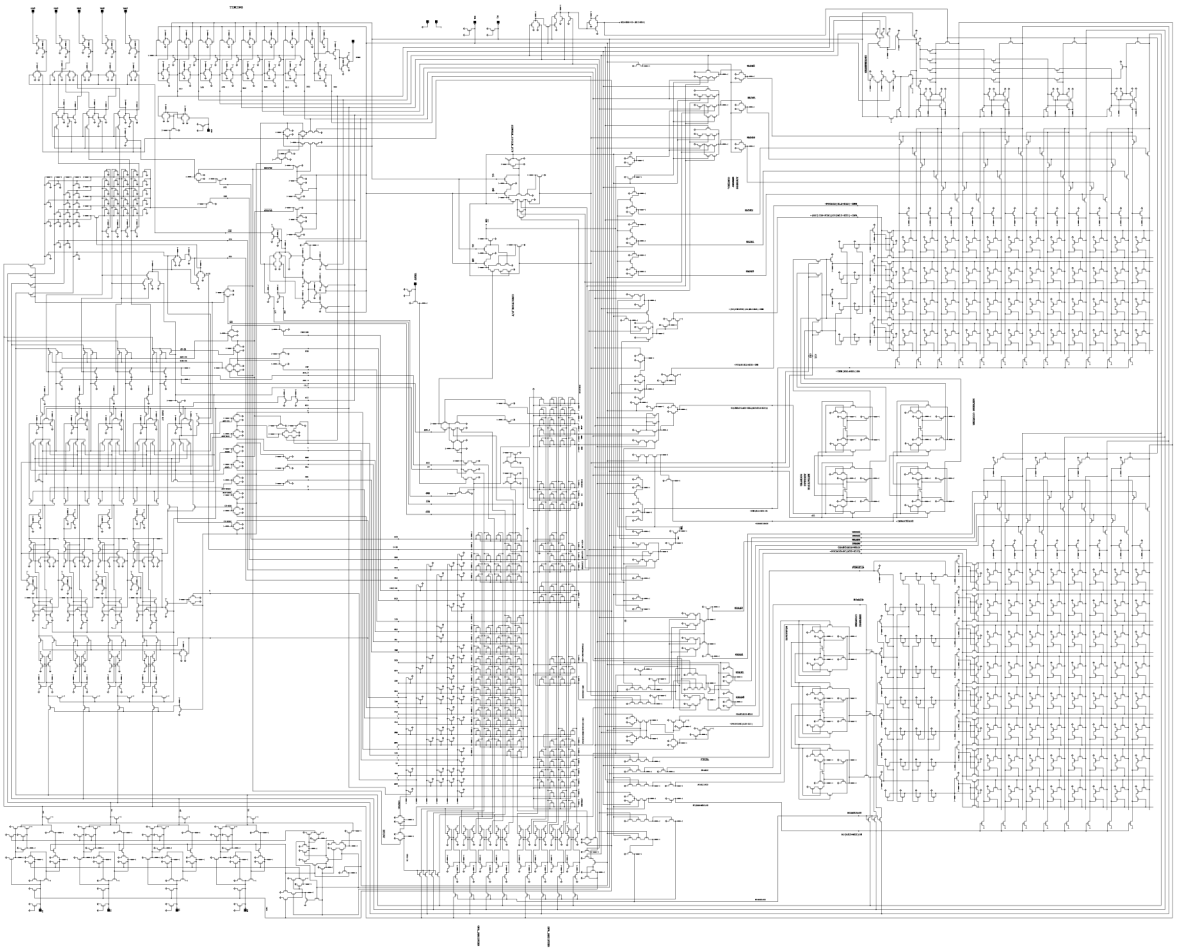


Рисунок 1.1. Цифровая схема процессора intel 4004 [1].

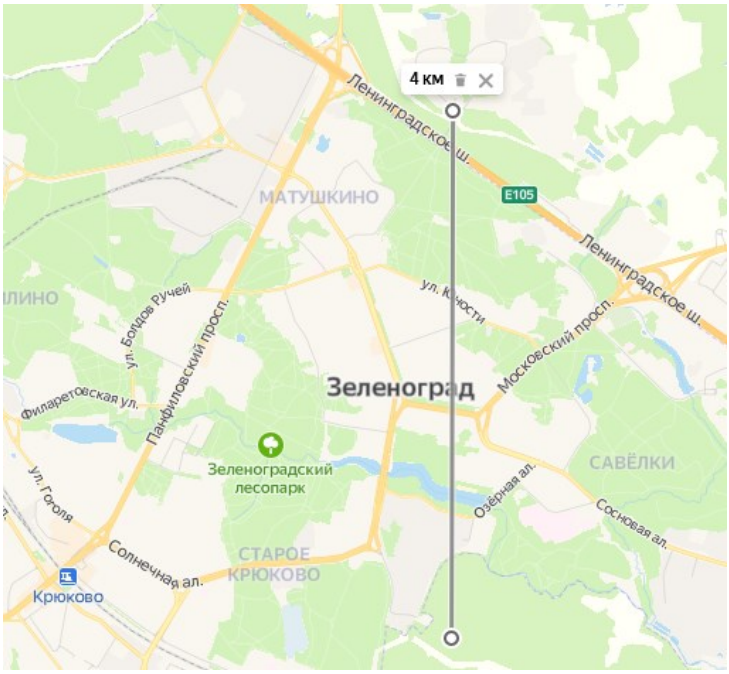


Рисунок 1.2. Демонстрация площади, которую могли бы занимать цифровые схемы современных процессоров

Как вы можете догадаться в какой-то момент между 1971 -ым и 2022-ым годами инженеры перестали разрабатывать цифровые схемы, рисуя их на бумаге. Разумеется, разрабатывая устройство, не обязательно вырисовывать на схеме каждый транзистор — можно управлять сложностью, переходя с одного уровня абстракции на другой. Например начинать разработку схемы с соединения функциональных блоков, а затем рисовать схему для каждого отдельного блока. К примеру, схему intel 4004 можно представить в следующем виде:

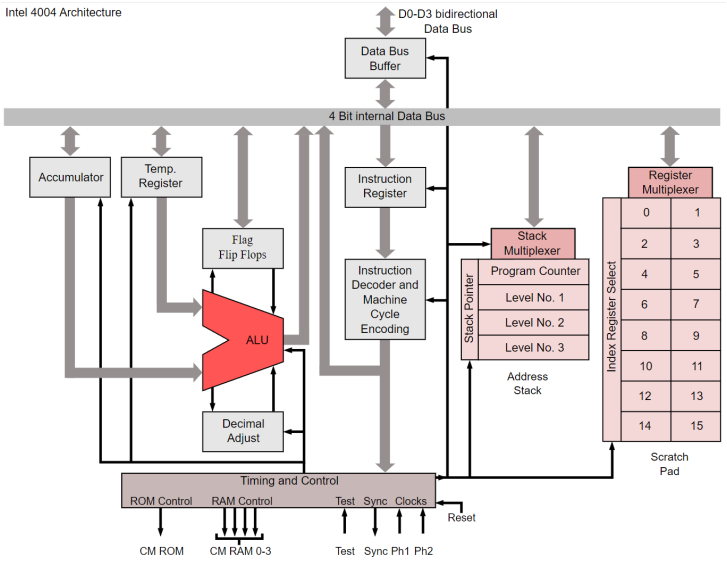


Рисунок 1.3. Структурная схема intel 4004[2]

Однако несмотря на это, даже отдельные блоки порой бывают довольно сложны. Возьмем блок аппаратного шифрования по алгоритму AES на рисунке 4:

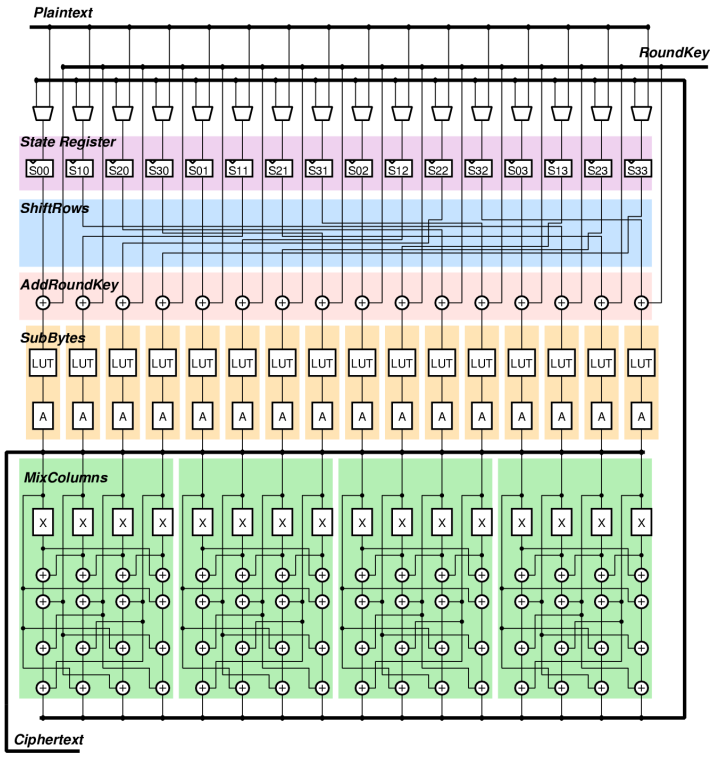


Рисунок 1.4. Структурная схема блока аппаратного шифрования по алгоритму AES[3]

Заметьте, что даже этот блок не является атомарным. Каждая операция Исключающего ИЛИ, умножения, мультиплексирования сигнала и таблицы подстановки — это отдельные блоки, функционал которых еще надо реализовать.

В какой-то момент, инженеры поняли что проще описать цифровую схему в тексовом представлении, нежели в графическом.
Как можно описать цифровую схему текстом? Рассмотрим цифровую схему полусумматора:

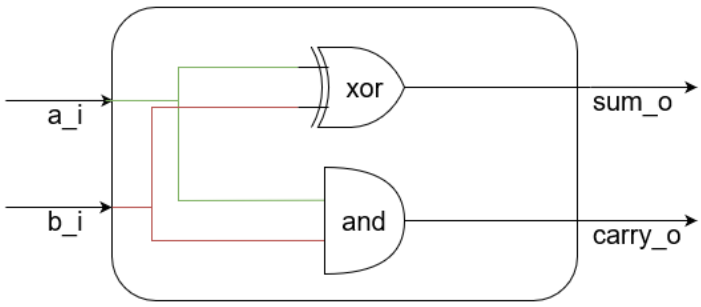


Рисунок 1.5. Цифровая схема полусумматора.

Это **устройство** (*полусумматор*) имеет два **входа**: *a* и *b*, а так же два **выхода**: *S* и *P*. Выход *S* является **результатом** логической операции **Исключающее ИЛИ** от операндов *a* и *b*. Выход *P* является **результатом** логической операции **И** от операндов *a* и *b*.

Текст выше и является тем описанием, по которому можно воссоздать эту цифровую схему. Если стандартизировать описание схемы, то в нем можно будет оставить только слова, выделенные жирным и курсивом. Пример того, как можно было бы описать эту схему по стандарту IEEE 1364-2005 (язык описания аппаратуры (Hardware Description Language — HDL) Verilog):

```
module half_sum(     // устройство полусумматор со
  input   a,         // входом a,
  input   b,         // входом b,
  output  S,         // выходом S и
  output  P          // выходом P.
);


assign S = a ^ b;
// Где выход S является
//результатом Исключающего ИЛИ от a и b,


assign P = a & b;    // а выход P является результатом
                     //логического И от a и b.


endmodule
```

На первый взгляд кажется, что такое описание даже больше, чем записанное естественным языком, однако так кажется только из-за переноса строк и некоторой избыточности в описании входов и выходов, которая была добавлена для повышения читаемости. То же самое описание можно было записать и в виде:

```
module half_sum(input a, b, output S, P);
  assign S = a ^ b;
  assign P = a & b;
endmodule
```

Обратите внимание, что код на языке Verilog описывает устройство целиком, одномоментно. Это описание схемы выше, а не построчное выполнение программы. Может показаться, что описывать устройство текстом сложнее, чем рисовать схему, тем более, что сперва мы **уже нарисовали схему**, а затем её описали. Однако, с практикой описание схемы в текстовом виде становится намного проще и не требует диаграммы. Для описания достаточно только спецификации: формальной записи того, что должно делать устройство, по которой разрабатывается алгоритм, который затем претворяется в описание на HDL.

Занятный факт: ранее было высказано предположение о том, что инженеры перестали разрабатывать устройства, рисуя цифровые схемы в промежуток времени между 1971-ым и 2022-ым годами. Так вот, первая конференция, посвященная языкам описания аппаратуры состоялась в 1973-ем году[4]. Таким образом, Intel 4004 можно считать одним из последних цифровых устройств, разработанных без использования языков описания аппаратуры.

2. История появления ПЛИС

До появления интегральных схем, электронные схемы собирались из отдельных элементов, как модель, собранная из кубиков Lego. В случае, если при сборке электронной схемы была допущена ошибка, вы могли исправить её ручной корректировкой соединения элементов подобно исправлению ошибки, допущенной при сборке модели Lego.
С улучшением технологических процессов произошла миниатюризация базовых элементов, из которых состоят электронные схемы, что привело к появлению интегральных схем — электронных схем, выполненных на полупроводниковой подложке и заключенных в неразборный корпус.
В большинстве случаев, исправить ошибку, допущенную при разработке и изготовлении интегральной схемы становится невозможным. С учетом того, что изготовление прототипа интегральной схемы является затратным мероприятием (от десятков тысяч до миллионов долларов в зависимости от тех процесса и площади изготавливаемого кристалла), возникла необходимость в способе проверки прототипа схемы до изготовления её прототипа. Так появились **программируемые логические интегральные схемы** (**ПЛИС**).
ПЛИС содержит некое конечное множество базовых блоков (примитивов), программируемые блоки межсоединений примитивов и блоки ввода-вывода. Подав определенный набор воздействий на ПЛИС (**запрограммировав** её), можно настроить примитивы, их межсоединения между собой и блоками ввода-вывода, чтобы получить определенную цифровую схему.
Удобство ПЛИС заключается в том, что в случае обнаружения ошибки на прототипе, исполненном в ПЛИС, вы можете исправить свою цифровую схему, и повторно запрограммировать ПЛИС.
Кроме того, эффективно использовать ПЛИС не как средство дешевого прототипирования, но и как средство реализации конечного продукта в случае малого тиража (дешевле купить и запрограммировать готовую партию ПЛИС, чем изготовить партию собственных микросхем).

Стоит оговориться, что обычно под термином ПЛИС подразумевается конкретный тип программируемых схем: **FPGA** (**field-programmable gate array**, **программируемая пользователем вентильная матрица**) здесь и далее термин **ПЛИС** будет использоваться как синоним **FPGA**.

Давайте разберемся что же это за устройство и как оно работает изнутри, но перед этим необходимо провести ликбез по цифровым схемам и логическим вентилям.

Цифровые схемы

В электронике, словом "цифровая" описывают схемы, которые абстрагируются от непрерывных (аналоговых) значений напряжений, вместо этого используется только два дискретных значения: `0` и `1`. На текущем уровне абстракции нас не интересуют конкретные значения напряжений и пороги этих значений. К примеру в ПЛИС часто используются значения `1.2 В` в качестве `1` и `0 В` в качестве `0`. Если реальным значением сигнала будет напряжение `0.8 В`, что достаточно близко к `1.2 В`, оно всё ещё будет считаться `1`. Цифровые схемы разрабатываются таким образом, чтобы устанавливать крайние значения напряжений (сигнал, имеющий значение напряжения `0.8 В`, пройдя через очередной логический вентиль будет иметь значение напряжения около `1.2 В`), что делает их крайне устойчивыми к шумам и влиянию внешнего мира. Таким образом, концепция "цифровой схемы" позволяет нам уйти от всего этого сложного поведения на уровне напряжений, давая нам возможность разрабатывать схему в идеальном мире, где у напряжения может быть всего два значения: `0` и `1`. А обеспечением этих условий будут заниматься базовые блоки, из которых мы будем строить цифровые схемы.

Эти базовые блоки называются **логическими вентилями**.

Логические вентили

Существует множество логических вентилей, но чаще всего используется четыре из них: **И**, **ИЛИ**, **Исключающее ИЛИ**, **НЕ**. Каждый из этих элементов принимает на вход **цифровое значение** (см. **цифровая схема**), выполняет определенную **логическую функцию** над входами и подает на выход результат этой функции в виде **цифрового значения**.

Логический вентиль **И** принимает два входа и выдает на выход значение `1` только в том случае, если оба входа равны `1`. Если хотя бы один из входов `0`, то на выходе будет `0`. На схемах, логический вентиль **И** отображается следующим образом:

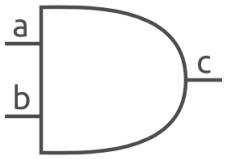


Рисунок 2.1. Обозначение логического вентиля **И**

Логический вентиль **ИЛИ** принимает два входа и выдает на выход значение `1` в случае, если хотя бы один из входов равен `1`. Если оба входа равны `0`, то на выходе будет `0`. На схемах, логический вентиль **ИЛИ** отображается следующим образом:

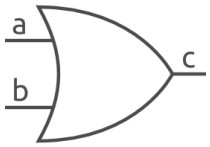


Рисунок 2.2. Обозначение логического вентиля **ИЛИ**

Логический вентиль **Исключающее ИЛИ** принимает два входа и выдает на выход значение `1` в случае, если значения входов не равны между собой (один из них равен `1`, а другой `0`). Если значения входов равны между собой (оба равны `0` или оба равны `1`), то на выходе будет `1`. На схемах, логический вентиль **Исключающее ИЛИ** отображается следующим образом:

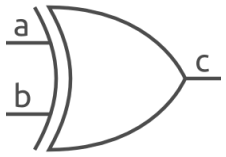


Рисунок 2.3. Обозначение логического вентиля **Исключающее ИЛИ**

Логический вентиль **НЕ** является самым простым вентилем. Он принимает один вход и подает на выход его инверсию. Если на вход пришло значение 0, то на выходе будет 1, если на вход пришло значение 1, то на выходе будет 0. На схемах, логический вентиль **НЕ** отображается следующим образом:

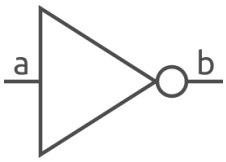


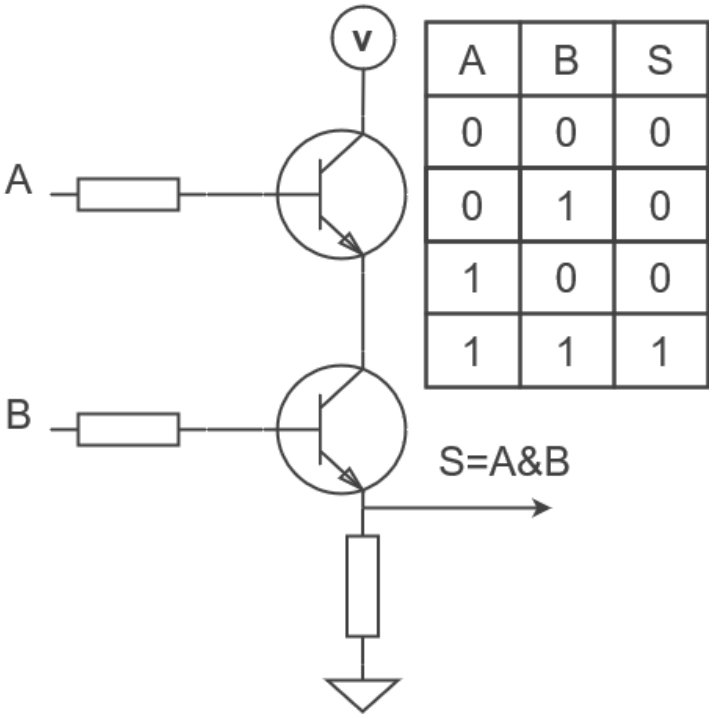
Рисунок 2.4. Обозначение логического вентиля **НЕ**

Так же существуют вариации базовых вентилей, такие как **И-НЕ**, **ИЛИ-НЕ**, **Исключающее ИЛИ-НЕ**, отличающиеся от исходных тем, что их выходы инвертируются.

Логические вентили строятся из **транзисторов**. **Транзистор** — это полупроводниковый элемент, может

пропускать/блокировать ток в зависимости от поданного напряжения на его управляющий вход.

На приведенном ниже рисунке показан способ построения логического вентиля **И** на базе двух транзисторов. Подача значения `1` на вход **A** или **B** "открывает" соответствующий транзистор. Если оба транзистора открыты, на выход идет **напряжение питания** (`1` в контексте **цифровых значений**). В случае, если хотя бы на одном входе **A** или **B** будет значение `0`, соответствующий транзистор будет закрыт (можно считать что он превратится в разрыв цепи). В этом случае выход будет подключен к **земле** (`0` в контексте цифровых значений). Как вы видите, напряжение на выход подается от **источников постоянного питания** или **земли**, а не от входов вентиля, именно этим и обеспечивается постоянное обновление напряжения и устойчивость **цифровых схем** к помехам.



A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

Рисунок 2.5. Обозначение логического вентиля **Схема логического вентиля И, построенного на транзисторах**

Теперь, имея базовое представление о транзисторах и логических вентилях, мы можем построить из них что-то полезное. Используя одни лишь описанные выше логические вентили можно построить **любую(!)** цифровую схему.

Однако, при описаний цифровых схем, некоторые цифровые блоки используются настолько часто, что для них ввели отдельные символы (**сумматоры**, **умножители**, **мультиплексоры**), используемые при описании более сложных схем. Мы рассмотрим один из фундаментальных строительных блоков в ПЛИС — **мультиплексор**.

Мультиплексоры

Мультиплексор — это устройство, которое в зависимости от значения **управляющего сигнала** подает на выход значение одного из входных сигналов.

Схематически, мультиплексор обозначается следующим образом:

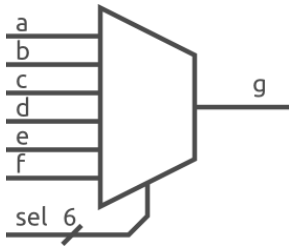


Рисунок 2.6. Обозначение Мультиплексора

Символ `/` на линии `sel` используется, чтобы показать что этот сигнал шириной 6 бит.

Число входов мультиплексора может быть различным, но выход у него всегда один.

Способ, которым кодируется значение управляющего сигнала может также различаться. Простейшая цифровая схема мультиплексора получится, если использовать `one-hot`-кодирование. При таком кодировании, значение **многоразрядного** сигнала **всегда** содержит **ровно одну** `1`. Информация, которую несет закодированный таким образом сигнал содержится в положении этой `1` внутри многоразрядного сигнала.

Посмотрим, как можно реализовать мультиплексор с управляющим сигналом, использующим `one-hot`-кодирование, используя только логические вентили **И**, **ИЛИ**:

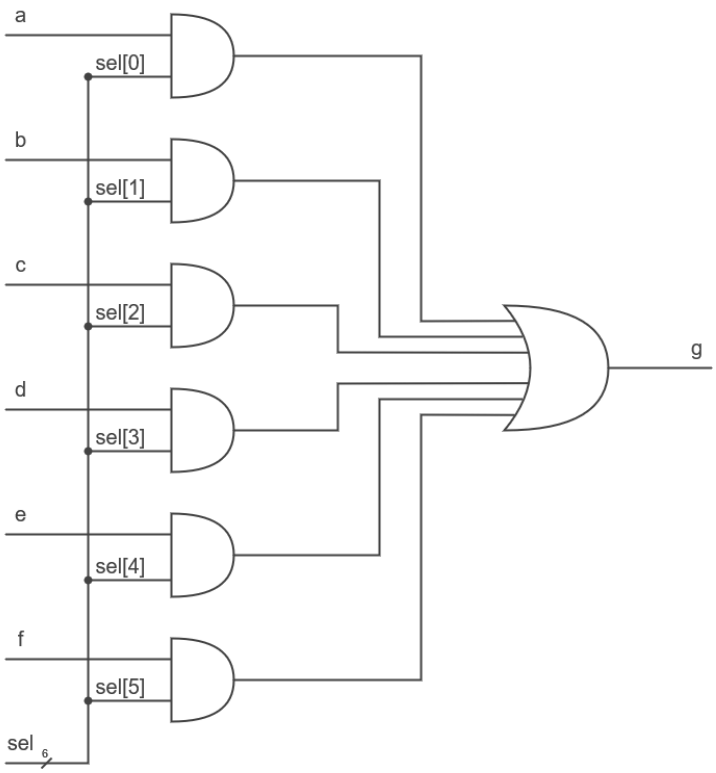


Рисунок 2.7. Реализация мультиплексора, использующего one-hot кодирование

Если мы выставим значение управляющего сигнала, равное `000010`, означающее что только **первый** бит этого сигнала (**счет ведется с нуля**) будет равен **единице** (`sel[1] = 1`), то увидим, что на один из входов каждого логического вентиля **И** будет подано значение `0`. Исключением будет логический вентиль **И** для входа `b`, на вход которого будет подано значение `1`. Это означает, что все логические вентили **И** (кроме первого, на который подается вход `b`) будут выдавать на выход `0` (см. Логические вентили) вне зависимости от того, что было подано на входы a,c,d,e и f. Единственным входом, который будет на что-то влиять окажется вход `b`. Когда он равен `1`, на выходе соответствующего логического вентиля **И** окажется значение `1`. Когда он равен `0` на выходе **И** окажется значение `0`. Иными словами, выход **И** будет повторять значение `b`.

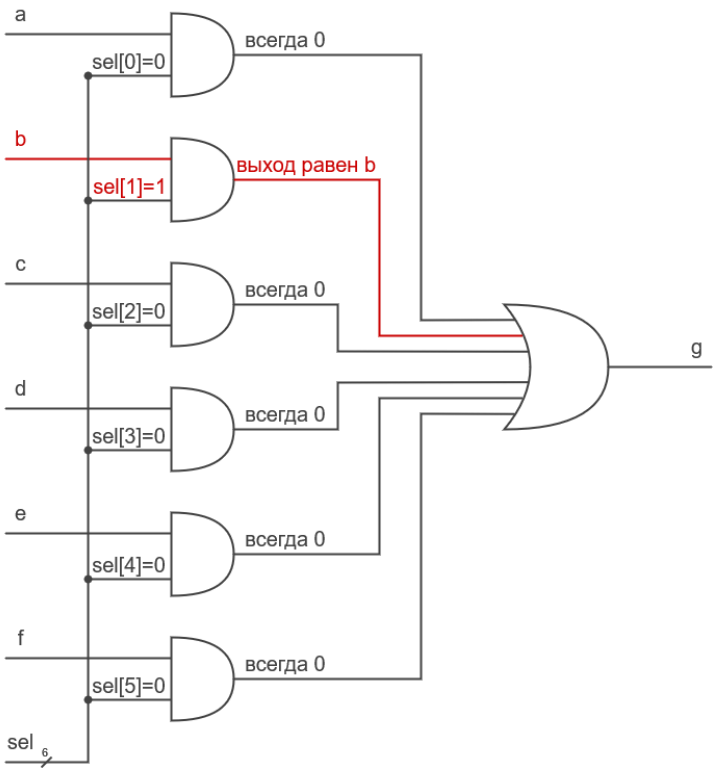


Рисунок 2.8. Реализация мультиплексора, использующего one-hot кодирование

Логический вентиль **ИЛИ** на данной схеме имеет больше двух входов. Подобный вентиль может быть создан в виде каскада логических вентилей **ИЛИ**:

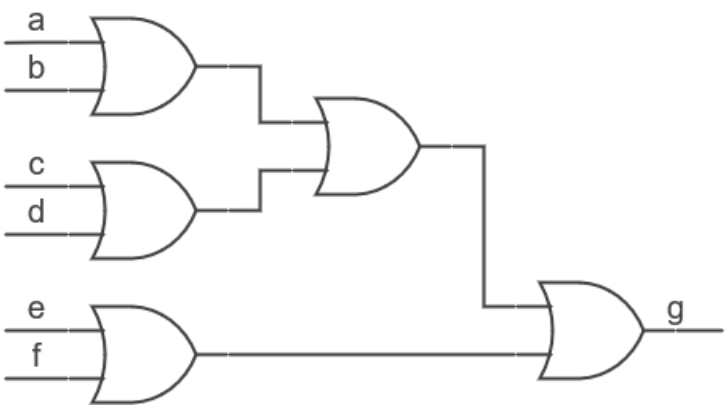


Рисунок 2.9. Реализация многоходового логического **ИЛИ**

Многовходовой вентиль ИЛИ ведет себя ровно так же, как двухвходовой: он выдает на выход значение `1` когда хотя бы один из входов равен `1`. В случае, если все входы равны `0`, на выход **ИЛИ** пойдет `0`.

Но для нашей схемы мультиплексора гарантируется, что каждый вход **ИЛИ** кроме одного будет равняться 0 (поскольку выход каждого **И** кроме одного будет равен 0). Это означает, что выход **многовходового ИЛИ** будет зависеть только от **одного** входа (в случае, когда sel = 000010 — от входа b).

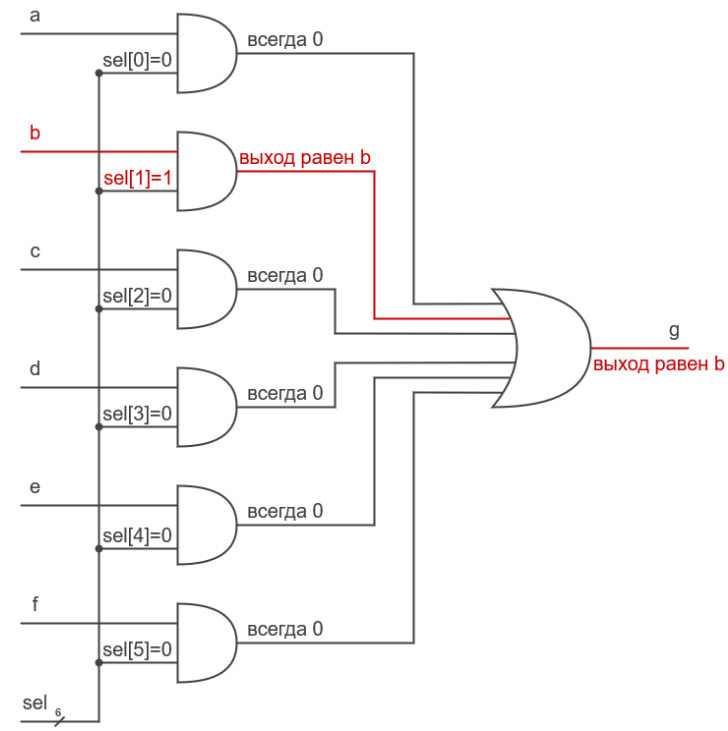


Рисунок 2.10. Реализация мультиплексора, использующего one-hot кодирование

Меняя значение sel, мы можем управлять тем, какой из входов мультиплексора будет управлять его выходом.

Теперь, попробуйте представить огромную матрицу мультиплексоров, у которых можно "запрограммировать" управляющий сигнал sel (под "запрограммировать" подразумевается "выставить то значение, которое нам нужно"). Это позволит направлять сигналы вашей цифровой схемы туда, куда вам будет нужно. Именно так ПЛИС и управляет тем, куда именно приходят сигналы.

Разумеется, **маршрутизация миллионов сигналов** — дело запутанное, но по своей сути **это всего лишь куча мультиплексоров, у которых управляющий сигнал sel подключен к программируемой памяти**.

Таблицы подстановки (Look-Up Tables, LUTs)

Итак, у нас есть способ динамически менять маршрут сигналов и приводить их туда, куда нам нужно. Теперь необходимо понять, как генерировать произвольную логику. И для этого мы снова воспользуемся мультиплексорами, в частности их производными, которые называются **Таблицы подстановки** или **Look-Up Tables** (**LUTs**).

Представьте мультиплексор с четырьмя входными сигналами, и двухбитным управляющим сигналом (обратите внимание, что в теперь это сигнал не использует `one-hot`-кодирование). Но теперь, вместо того, чтобы выставлять входные сигналы во внешний мир, давайте подключим их к программируемой памяти. Это означает, что мы можем "запрограммировать" каждый из входов на какое-то константное значение. Поместим то что у нас получилось в отдельный блок и вот, мы получили двухвходовую **Таблицу подстановки** (далее **LUT**).

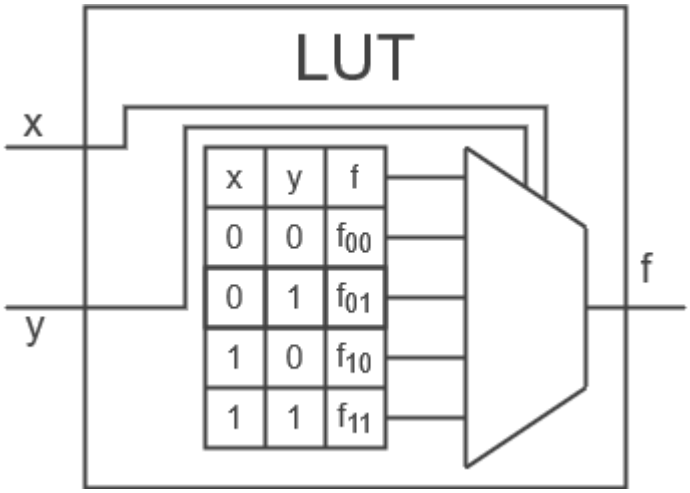


Рисунок 2.11. Реализация таблицы подстановки (Look-Up Table, LUT)

Эти два входа **LUT** являются битами управляющего сигнала мультиплексора, спрятанного внутри **LUT**. Программируя входы мультиплексора (точнее, программируя память, к которой подключены входы мультиплексора), мы можем получить из **LUT** **любую (!)** логическую функцию, принимающую два входа и возвращающую один выход.

Допустим мы хотим получить **логическое И**. Для этого, нам потребуется записать в память следующее содержимое:

Адрес (In[1:0])	Значение
00	0
01	0
10	0
11	1

Это простейший пример — обычно **LUT**-ы имеют больше входов, что позволяет им реализовывать более сложную логику.

D-триггеры

Как вы уже поняли, используя неограниченное количество LUT-ов, вы можете построить цифровую схему, реализующую логическую функцию любой сложности. Однако цифровые схемы не ограничиваются реализацией одних только логических функций (цифровые схемы, реализующие логическую функцию называются **комбинационными**, поскольку выход зависит только от комбинации входов). Например, так не построить цифровую схему, реализующую процессор. Для таких схем, нужны элементы памяти. Заметим, что речь идет не о программируемой памяти, задавая значения которой мы управляем тем, куда будут направлены сигналы, и какие логические функции будут реализовывать LUT-ы. Речь идет о ячейках памяти, которые будут использоваться логикой самой схемы. Такой базовой ячейкой памяти является **D-триггер**, из которых можно собрать другие ячейки памяти, например **регистры** (а из регистров можно собрать **память с произвольным доступом** (**random access memory**, **RAM**)), **сдвиговые регистры** и т.п.

D-триггер — это цифровой элемент, способный хранить один бит информации. В базовом варианте у этого элемента есть два входа и один выход. Один из входов подает значение, которое будет записано в **D-триггер**, второй вход управляет записью (обычно он называется clk или clock и подключается к тактирующему синхроимпульсу схемы). Когда управляющий сигнал меняет свое значение с 0 на 1 (либо с 1 на 0, зависит от схемы), в **D-триггер** записывается значение сигнала данных. Обычно, описывая **D-триггер**, говорится, что он строится из двух защелок, которые в свою очередь строятся из **RS-триггеров**, однако в конечном итоге, все эти элементы строятся на базе логических вентилей **И**/**ИЛИ**, **НЕ**:

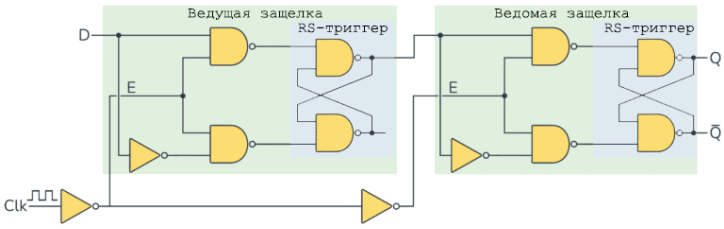


Рисунок 2.12. Реализация D-триггера

Арифметика

Помимо описанных выше блоков (мультиплексоров и построенных на их основе LUT-ов и регистров) выделяется еще один тип блоков, настолько часто используемый в цифровых схемах, что его заранее размещают в ПЛИС в больших количествах: это арифметические блоки. Эти блоки используются при сложении, вычитании, сравнении чисел, реализации счётчиков. В разных ПЛИС могут быть предустановлены разные блоки: где-то это может быть однобитный сумматор, а где-то блок вычисления ускоренного переноса (carry-chain).

Все эти блоки могут быть реализованы через логические вентили, например так можно реализовать сумматор:

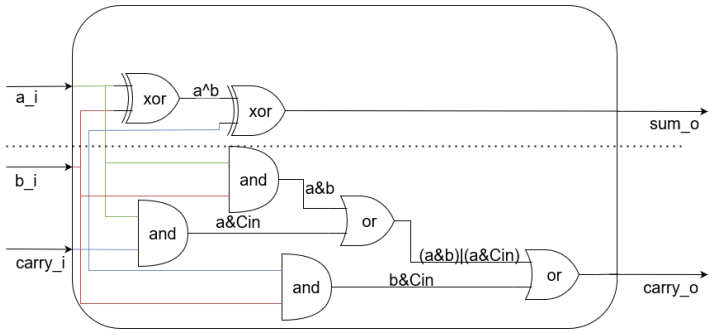


Рисунок 2.13. Реализация полного однобитного сумматора

Логическая ячейка

И вот, мы подходим к внутреннему устройству ПЛИС. Мы уже узнали, что в ПЛИС есть матрица программируемых мультиплексоров, направляющих сигналы туда, куда нам нужно.
Вторым важным элементом является **логический блок** (обычно состоящих из **логических ячеек**, но для простоты мы отождествим эти два термина).

Логический блок содержит одну или несколько **LUT**, **арифметический блок**, и один или несколько **D-триггеров**, которые соединены между собой некоторым количеством мультиплексоров. Ниже представлена схема того, как может выглядеть **логический блок**:

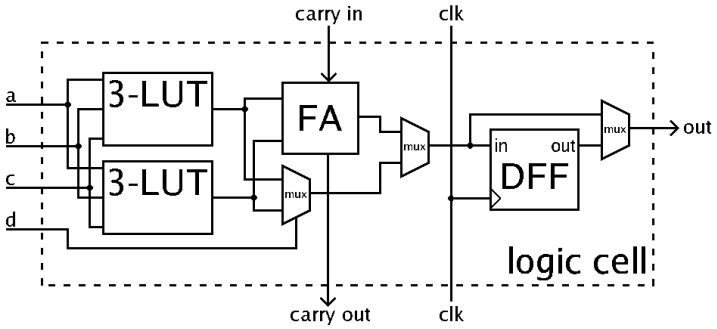


Рисунок 2.14. Схема логической ячейки[3]

Может показаться запутанным, но все достаточно просто. Логический блок представляет собой цепочку операций: логическая функция, реализованная через LUT -> арифметическая операция -> Запись в D-триггер. Каждый из мультиплексоров определяет то, будет ли

пропущен какой-либо из этих этапов. Таким образом, конфигурируя каждый логический блок, можно получить следующие вариации кусочка цифровой схемы:

1. Комбинационная схема (логическая функция, реализованная в LUT)
2. Арифметическая операция
3. Запись данных в D-триггер
4. Комбинационная схема, с записью результата в D-триггер
5. Арифметическая операция с записью результата в D-триггер
6. Комбинационная схема с последующей арифметической операцией
7. Комбинационная схема с последующей арифметической операцией и записью в D-триггер

А вот реальный пример использования логического блока в ПЛИС xc7a100tcsg324-1 при реализации Арифметико-логического устройства (АЛУ), подключенного к периферии отладочной платы Nexys-7:

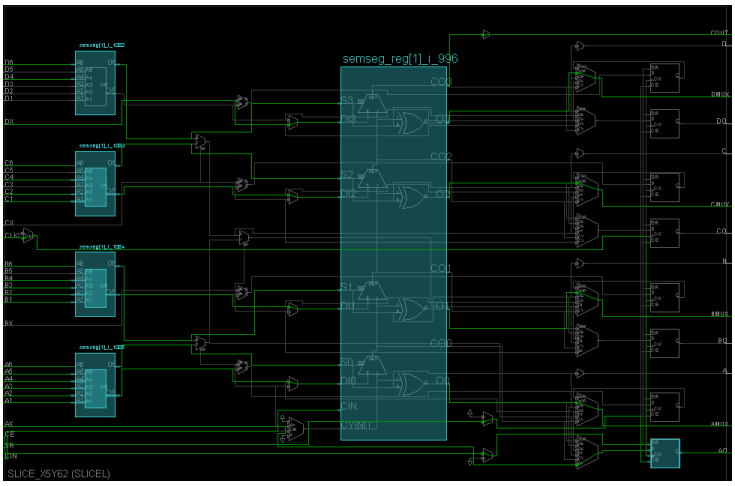


Рисунок 2.15. Пример использования логической ячейки

Здесь вы можете увидеть использование LUT-ов, блока расчета ускоренного переноса, и одного из D-триггеров. D -триггеры, обозначенные серым цветом, не используются.

Располагая большим наборов таких логических блоков, и имея возможность межсоединять их нужным вам образом, вы получаете широчайшие возможности по реализации практически любой цифровой схемы (ограничением является только ёмкость ПЛИС, т.е. количество подобных логических блоков, входов выходов и т.п.).

Помимо логических блоков, в ПЛИС есть и другие примитивы: блочная память, блоки умножителей и т.п.

Обобщим сказанное:

Используя такие полупроводниковые элементы, как транзисторы, можно собирать логические вентили: элементы **И**, **ИЛИ**, **НЕ** и т.п.

Используя **логические вентили**, можно создавать схемы, реализующие как **логические функции** (**комбинационные схемы**), так и сложную логику с памятью (**синхронные схемы**).

Из логических вентилей среди прочего строится и такая важная комбинационная схема, как **мультиплексор**: цифровой блок, в зависимости от управляющего сигнала подающий на выход один из входных сигналов.

Подключив управляющий сигнал мультиплексора к **программируемой памяти** можно управлять тем, какие сигналы пойдут на выход и направлять их в нужную часть схемы (**маршрутизировать сигналы**).

Подключив входные сигналы мультиплексора к программируемой памяти, можно получить **Таблицу подстановок** (**Look-Up Table**, **LUT**), которая может реализовывать простейшие логические функции. LUT-ы позволяют заменить логические вентили И/ИЛИ/НЕ, и удобны тем, что их можно динамически изменять, логические вентили в свою очередь исполняются на заводе и уже не могут быть изменены после создания.

Из логических вентилей так же можно собрать базовую ячейку памяти: **D-триггер**, и такую часто используемую комбинационную схему как **полный однобитный сумматор** (или любой другой часто используемый арифметический блок).

Объединив LUT, арифметический блок и D-триггер получается структура в ПЛИС, которая называется **логический блок**.

Логический блок (а так же другие **примитивы**, такие как **блочная память** или **умножители**) — это множество блоков, которые заранее физически размещаются в кристалле ПЛИС, их количество строго определено конкретной ПЛИС и не может быть изменено.

Конфигурируя примитивы и **маршрутизируя сигнал** между ними (см. п.4), можно получить **практически любую цифровую схему** (с учетом ограничения ёмкости ПЛИС).

3. Шаги реализации разработанного устройства в ПЛИС

Для того, чтобы описанное на **языке описания аппаратуры** устройство было реализовано в ПЛИС, необходимо выполнить несколько этапов:

1. Элаборацию (elaboration)
2. Синтез (synthesis)
3. Имплементацию (implementation)
4. Генерацию битстрима (generate bitstream)
5. Остановимся на каждом шаге подробнее:

Elaboration

На шаге **элаборации**, САПР строит цифровую схему по её **HDL**-описанию. Построенная схема не привязана к конкретной ПЛИС и использует абстрактные элементы: логические вентили, мультиплексоры, элементы памяти и т.п. На самом деле САПР генерирует не схему (картинку, схематик), а так называемый **нетлист**. **Нетлист** — это представление цифровой схемы в виде **графа**, где каждый элемент схемы является **узлом**, а **цепи**, соединяющие эти элементы являются ребрами. **Нетлист** может храниться как в виде каких-то внутренних файлов САПР-а (в случае **нетлиста** для этапа **элаборации**), так и в виде **HDL**-файла, описывающего экземпляры примитивов и связи между ними. Рассмотрим этап **элаборации** и термин **нетлиста** на примере.

Допустим, мы хотим реализовать следующую цифровую схему:

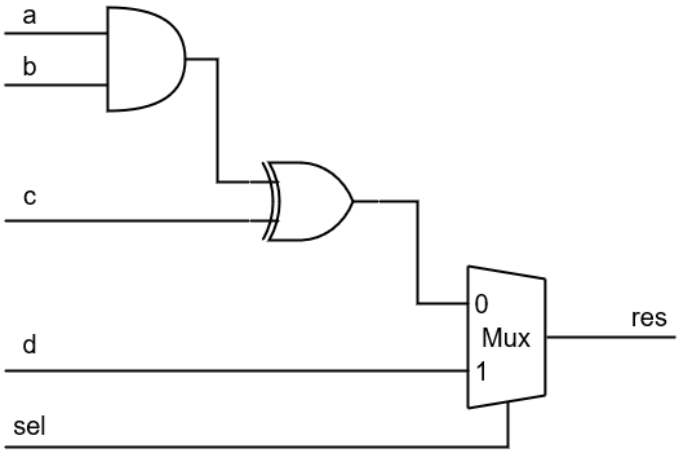


Рисунок 3.1. Пример разрабатываемой схемы

Её можно описать следующим **SystemVerilog**-кодом:

```
module sample(
    input logic a, b, c, d, sel,
    output logic res
    );

    logic ab = a & b;
    logic xabc = ab ^ c;

    assign res = sel? d : xabc;

endmodule
```

Выполним этап **элаборации**. Для этого в **Vivado** на вкладке RTL Analysis выберем Schematic.

Откроются следующие окна:

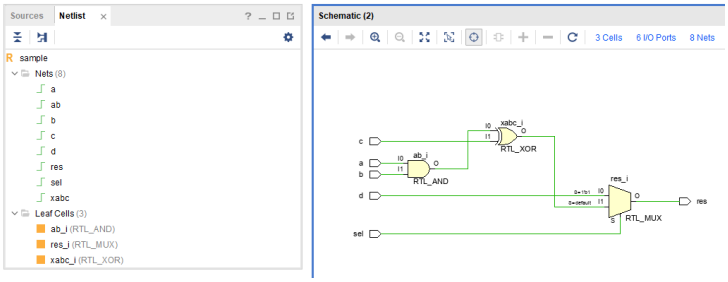


Рисунок 3.2. Схема после этапа элаборации

В левом окне мы видим наш нетлист. В нижней части обозначены узлы графа (элементы **ab_i**, **res_i**, **xabc_i**, которые представляют собой **И**, **мультиплексор** и **Исключающее ИЛИ** соответственно. Имена этих элементов схожи с именами проводов, присваиванием которым мы создавали данные элементы)

В верхней части обозначены **ребра графа**, соединяющие элементы схемы. Это входы и выходы нашего модуля, а так же созданные нами промежуточные цепи.

Справа вы видите **схематик** — **графическую схему**, построенную на основе данного **графа** (**нетлиста**).

Synthesis

На шаге синтеза, САПР берет сгенерированную ранее цифровую схему и реализует элементы этой схемы через примитивы конкретной ПЛИС — в основном через логические ячейки, содержащие таблицы подстановки, полный однобитный сумматор и D-триггер (см. как работает ПЛИС).

В рамках нашего примера, САПР смотрит на построенный на этапе **элаборации нетлист** и решает, какими средствами (**примитивами**) ПЛИС можно его реализовать. Поскольку схема чисто **комбинационная**, результат её работы можно рассчитать и выразить в виде **таблицы истинности**, а значит для её реализации лучше всего подойдут **Таблицы Подстановки** (**LUT**-ы). Более того, в ПЛИС xc7a100tcsg324-1 есть пятивходовые LUT-ы, а у нашей схемы именно столько входов. Это означает, работу всей этой схемы можно заменить **всего одной таблицей подстановки** внутри ПЛИС.

Итак, продолжим рассматривать наш пример и выполним этап синтеза. Для этого нажмем на кнопку Run Synthesis.

После выполнения синтеза у нас появится возможность открыть новый схематик, сделаем это.

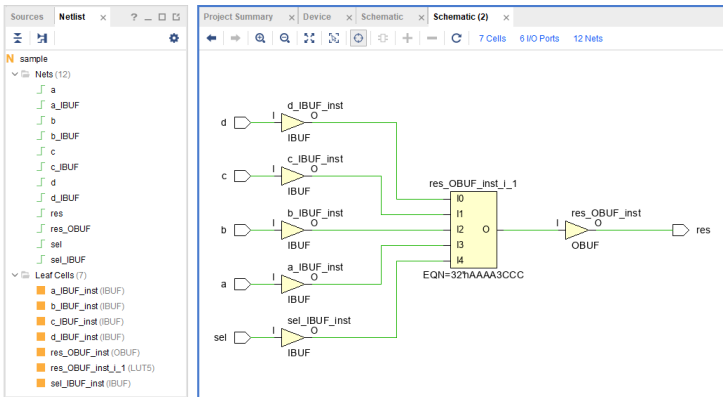


Рисунок 3.3. Схема после этапа синтеза

Мы видим, что между входами/выходами схемы и её внутренней логикой появились новые примитивы — **буферы**. Они нужны, преобразовать уровень напряжения между ножками ПЛИС и внутренней логикой (условно говоря, на вход плис могут приходить сигналы с уровнем 3.3 В, а внутри ПЛИС примитивы работают с сигналами уровня 1.2 В).

Сама же логика, как мы и предполагали, свернулась в одну пятивходовую таблицу подстановки.

строка EQN=32'hAAAA3CCC означает, что таблица подстановки будет инициализирована следующим 32-битным значением: 0xAAAA3CCC. Поскольку у схемы 5 входов, у нас может быть 2^5=32 возможных комбинаций входных сигналов и для каждого нужно свое значение результата.

Можно ли как-то проверить данное 32-битное значение без просчитывания всех 32 комбинаций сигналов "на бумажке"?

Да, можно.

Implementation

После построения новой схемы, где в качестве элементов используются ресурсы конкретной ПЛИС, происходит расчёт размещения этой схемы внутри ПЛИС: выбираются конкретные логические ячейки, происходит маршрутизация сигнала между этими ячейками. Например, реализация 32-битного сумматора с ускоренным переносом может потребовать 32 LUT-а и 8 примитивов вычисления быстрого переноса (CARRY4). Будет неразумно использовать для этого примитивы, разбросанные по всему кристаллу ПЛИС, ведь тогда придется выполнять сложную маршрутизацию сигнала, да и временные характеристики устройства так же пострадают (сигналу идущему от предыдущего разряда к следующему придется проходить больший путь). Вместо этого, САПР будет пытаться разместить схему таким образом, чтобы использовались близлежащие примитивы ПЛИС, для получения оптимальных характеристик.

Что именно считается "оптимальным" зависит от двух вещей: настроек САПР и **ограничений** (**constraints**), наложенных на имплементацию. Ограничения сужают область возможных решений по размещению примитивов внутри ПЛИС под определенные характеристики (временны́е и физические). Например, можно сказать, внутри ПЛИС схема должна быть размещена таким образом, чтобы время прохождения по **критическому пути** не превышало 20ns. Это временно́е ограничение. Так же нужно сообщить САПР, к какой ножке ПЛИС необходимо подключить входы и выходы нашей схемы — это физическое ограничение.

Ограничения описываются не на языке описания аппаратуры, вместо этого используются текстовые файлы специального формата, зависящего от конкретной САПР.

Пример используемых ограничений для лабораторной по АЛУ.

После выполнения имплементации, нетлист и схема остаются неизменными, однако использованные для реализации схемы примитивы получают свой "адрес" внутри ПЛИС:

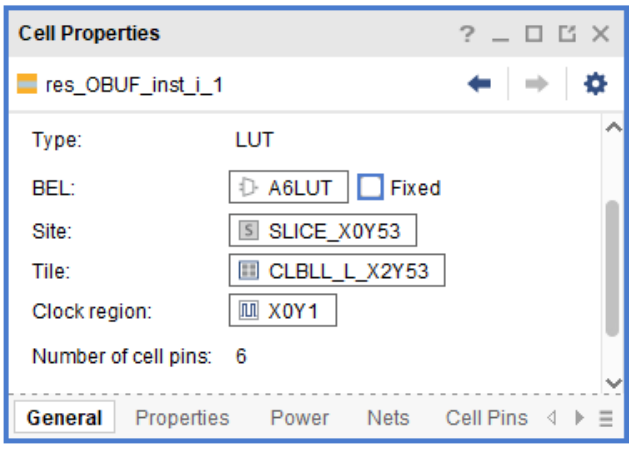


Рисунок 3.4. Свойства используемого примитива

Теперь, мы можем посмотреть на "внутренности" нашей ПЛИС xc7a100tcsg324-1 и то, как через её примитивы будет реализована наша схема. Для этого, необходимо отрыть имплементированное устройство: Implementation -> Open implemented design. Откроется следующее окно:

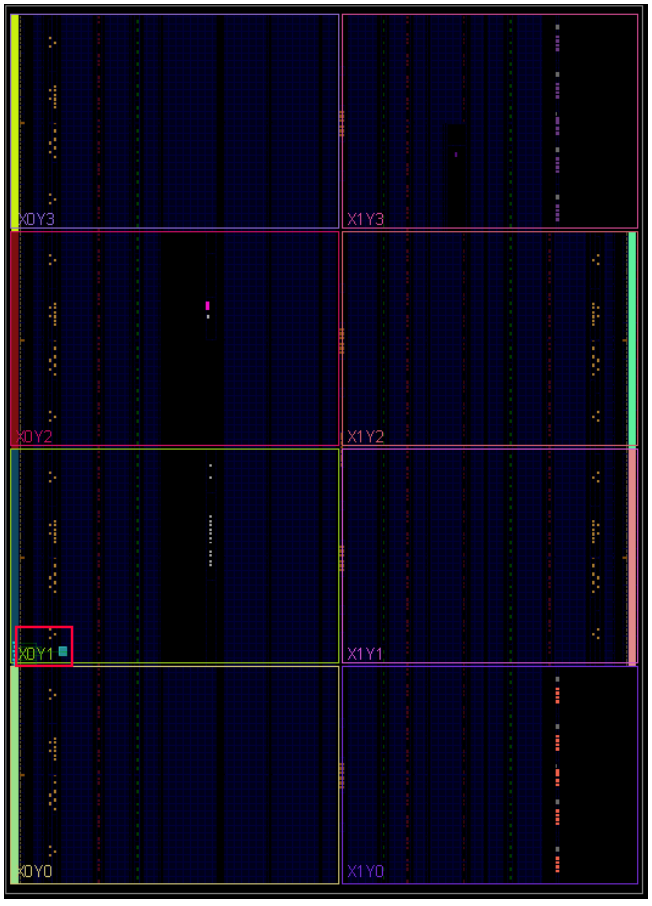


Рисунок 3.5. Результат имплементации

Может показаться очень страшным и непонятным, но это содержимое ПЛИС. Просто из-за огромного количества содержащихся в ней примитивов, она показана в таком масштабе, что все сливается в один цветной ковер. Большая часть этого окна неактивна (показана в темно-синих тонах) и это нормально, ведь мы реализовали крошечную цифровую схему, она и не должна занимать значительное количество ресурсов ПЛИС.

Нас интересует "бледно-голубая точка", расположенная в нижнем левом углу прямоугольника X0Y1 (выделено красным). Если отмасштабировать эту зону, мы найдем используемый нами LUT:

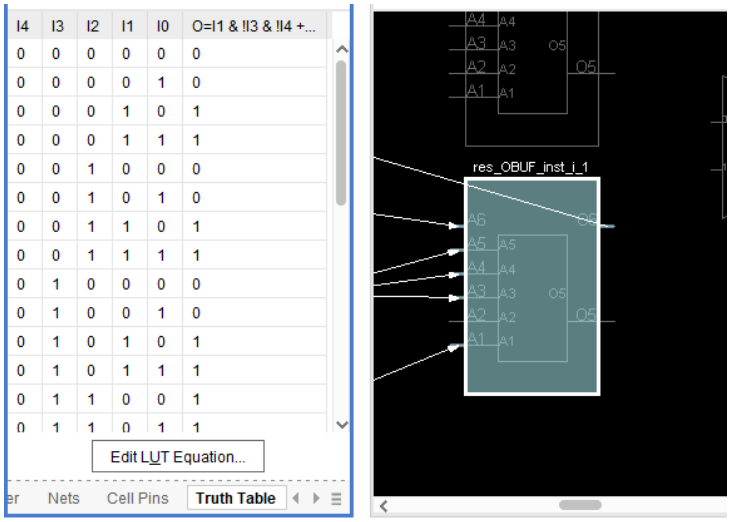


Рисунок 3.6. Используемая таблица подстановки (Look-Up Table, LUT)

Кроме того, если поиграться со свойствами этого примитива, мы сможем найти нашу таблицу истинности, инициализирующую этот примитив.

Generate Bitstream

После того, как САПР определил конкретные примитивы, их режим работы, и пути сигнала между ними, необходимо создать двоичный файл (**bitstream**), который позволит сконфигурировать ПЛИС необходимым нам образом. Получив этот файл, остается запрограммировать ПЛИС, после чего она воплотит разработанное устройство.

Выводы

Таким образом, маршрут перехода от HDL-описания устройства до его реализации в ПЛИС выглядит следующим образом:

Сперва происходит анализ HDL-описания. В ходе этого анализа выявляются простейшие структуры: регистры, мультиплексоры, вычислительные блоки (сложения/умножения/сдвига и т.п.). Строится граф схемы, построенной с помощью этих структур (**нетлист**). Данный нетлист платформонезависим, т.е. не привязан к конкретной ПЛИС.

После происходит этап синтеза нетлиста, полученного на

предыдущем этапе в нетлист, использующий имеющиеся ресурсы конкретной ПЛИС. Все, использовавшиеся на предыдущем этапе структуры (регистры, мультиплексоры и прочие блоки) реализуются через примитивы ПЛИС (LUT-ы, D-триггеры, блоки сложения и т.п.).

Затем происходит этап размещения схемы внутри ПЛИС: если на предыдущем этапе часть схемы была реализована через LUT, то на этом этапе решается **какой именно** LUT будет использован. Область допустимых решений по этому вопросу сужается путем наложения **ограничений** (**constraints**).

После размещения остается только сгенерировать двоичный файл (**bitstream**), который во время прошивки сконфигурирует ПЛИС на реализацию нашей схемы.

Источники

1. http://www.4004.com
2. Число транзисторов intel 4004, структурная схема intel 4004: https://en.wikipedia.org/wiki/Intel_4004
3. Структурная схема блока аппаратного шифрованиия по алгоритму AES: https://iis-people.ee.ethz.ch/~kgf/acacia/c3.html
4. Дата проведения первой конференции, посвященной HDL: "Verilog HDL and Its Ancestors and Descendants", стр. 87
5. Разделы 2, 3: How Does an FPGA Work?
6. Схема D-триггера: The D Flip-Flop (Quickstart Tutorial)
7. Схема логической ячейки: Field-programmable gate array

Зажигаем светодиод процессором J1

Балакший Сергей, *e-mail:* sergebn@mail.ru
Телеграм: @SergeBN

Обсуждение и комментарии :: ссылка

Аннотация

Статья для начинающих на тему “помигать светодиодом”. Приведен разбор того, как связаны между собой компилятор, система процессорных команд и “железо”, на котором все это работает.

1 Введение

Компилятор, система процессорных команд и “железо” тесно связаны между собой. В литературе обычно рассматривается система команд и проектирование процессора, на котором будет работать эта система команд. Вопрос связи получившейся конструкции (дизайна) с компилятором обычно находится на втором плане. В этой работе будет рассматриваться связь проектных решений, относящихся к конструкции процессора, с компилятором и программированием для разрабатываемого процессора. В качестве основной конструкции будет использован процессор J1 [4]. Это легко обозримый процессор. Исходный код процессора состоит из 130 строк, либо чуть больше, в зависимости от версии. Система команд J1 предназначена для реализации языка форт. Исходный код этого процессора доступен в [3].

Оригинальный процессор J1 [3] принадлежит Джеймсу Боумену. В данной работе этот процессор используется для наглядной демонстрации, чтобы не усложнять текст техническими деталями.

2 Основной дизайн J1

2.1 Функция определяет структуру

Процессор J1 относится к типу стековых процессоров. Он очень простой, но он имеет две шины данных. Одна шина - для данных. Другая шина - для команд. Это представляет некоторый интерес для проектирования в том смысле, что это процессор с Гарвардской [5] архитектурой, в котором канал инструкций и канал данных физически разделены. Рассмотрим описание инструкций и то, как функция, заданная в инструкции, будет определять структуру создаваемого дизайна процессора.

2.2 Краткое описание инструкций

В J1 предлагаются следующее формы для кодирования инструкций:

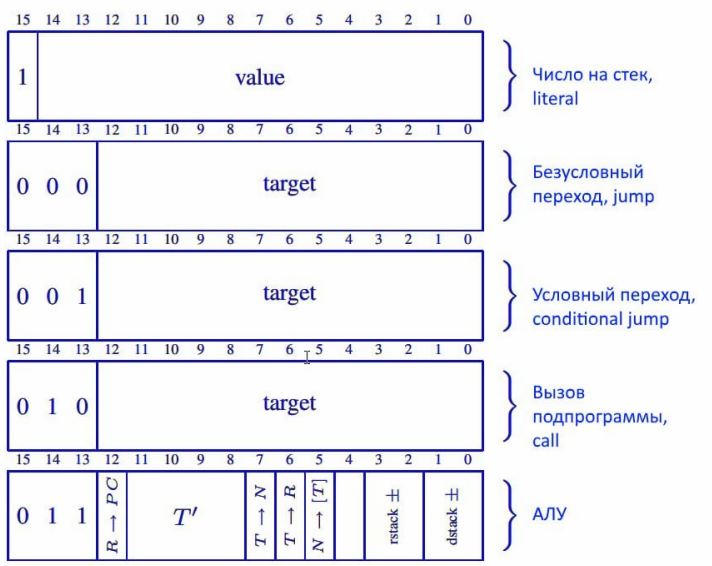


Рис. 1: Формат машинной команды

Перевод можно найти в [1]. Оригинал можно найти, как уже упоминалось, здесь [4].

Первая форма представляет собой описание литерала. Единица в старшем разряде определяет действия, которые необходимо будет выполнить, чтобы процессор правильно обработал данную форму. Следующие три формы определяют форматы команд: безусловный переход (J jump), условный переход (CJ conditional jump), вызов подпрограммы (CALL). Все три формы имеют поле для целевого адреса в 13 бит. Это накладывает ограничение на размер кода в 8К слов (16 битных инструкций) или в 16К байт.

Пятая форма кодирует операции АЛУ (арифметико логического устройства). Здесь в битах [11:8] кодируется

сама операция КОП (код операции). Биты [1:0] кодируют приращение для указателя на стек данных. Биты [3:2] кодируют приращение указателя стека возвратов. Биты [7:5] кодируют направление передачи данных. И бит 12 используется при выполнении команды возврата из подпрограммы. Бит 4 не используется вовсе и представляет интерес для дальнейших разработок. Хотя в реальном процессоре, который будет рассмотрен в этой статье, распределение бит немного отличается от презентации. Во всех формах три старших бита служат для декодирования этих форм.

2.3 Общие замечания

Итак, в начале было слово. Откуда же берется все остальное? Команда должна начинаться с какого-то действия, далее команда как-то должна исполняться, выполняя присущие ей действия, и потом команда должна на каком-то действии завершиться. То есть должна существовать некая среда, в которой и будут выполняться необходимые действия. Эта среда называется процессор. И единственная функция процессора - это вычислять.

Среда, в которой выполняется команда

Команды во время своего исполнения заставляют процессор производить некоторые действия. И каждую команду необходимо рассматривать в контексте того, как она воздействует на структуру - дизайн процессора. Причём именно функция, которую несёт в себе команда, и будет формировать, в определенной степени, окружение, в котором будет исполнятся команда. И конечно же, есть ещё разработчик, которому и принадлежит главенствующая роль при формировании такого окружения. Вообще говоря, одну и ту же функцию возможно реализовать, используя различные структуры. Например счеты, если кто-то помнит, что это такое, или узелковое письмо, или логарифмическая линейка, либо счетная машина “Феликс”. В распоряжении современного разработчика имеется такая структура как транзистор. И имея множество таких транзисторов, разработчик создает из них различные упорядоченные структуры, которые и выполняют то, что задумано. Очевидно, что невозможно создать что-то, что не задумано.

Интересно, как кто-то из читателей придумал бы своё окружение для выполнения этих команд?

Поскольку уже есть инструменты, при помощи которых становится возможным создавать эти самые упорядоченные структуры транзисторов. Один из таких инструментов называется verilog. Переходим непосредственно к дизайну.

2.4 Дизайн

Так как процессор стековый, то в нём должен присутствовать стек данных для работы с операндами, и стек возвратов, чтобы могла быть выполнена команда CALL.Также надо указывать на очередную команду, которая быть выпонена на следующем шаге. Для этого определяется счетчик команд, конечно, поскольку определены арифметические и логические операции, то должно присутствовать арифметико-логическое устройство (АЛУ, ALU).

Кроме того, при исполнении каждой инструкции, должны вырабатываться управляющие сигналы, которые будут осуществлять:

1. управление стеком данных;
2. управление стеком возвратов;
3. управление счетчиком команд.

То есть необходимо рассматривать такие вещи как:

1. Воздействие на вершину стека данных;
   - T’ обновить вершину стека новым или тем же значением;
   - T->N скопировать вершину стека;
   - T->R скопировать значение с вершины стека данных на вершину стека возвратов;
   - T->PC скопировать значение в счетчик команд;
   - N->[T] перенос значения в память;
   - +- измененить указателиь стека данных.
2. Воздействие на стек возвратов:
   - T->R скопировать значение с вершины стека данных на вершину стека возвратов;
   - R->T скопировать значение вершины стека возвратов на стек данных;
   - R->PC скопировать значение вершины стека возвратов в счетчик команд;
   - +- изменение указателя стека возвратов.
3. Воздействие на формирование сигналов управления памятью:
   - N->[T] перенос значения в память;
   - -[T] перенос значения из памяти на вершину стека.
4. Воздействие на формирование сигналов управления вводом выводом.

Проектирование процессоров имеет долгую историю [6]. Опираясь на этот опыт и уже установленные знания, и глядя на систему команд, можно предположить, что в состав процессора будут входить следующие вещи:

- Регистр вершины стека T (TOS Top of Stack).
- Регистр, следующий за вершиной стека N (Next).
- Стек данных dstack.
- Стек возвратов rstack.
- Указатель стека данных DSP.
- Указатель стека возвратов RSP.
- Регистр для счетчика команд.

Попробуем вывести структуру.

2.5 Счетчик команд (pc)

Формы 2-4 накладывают ограничения на счётчик команд в 13 разрядов. Следовательно, счётчик команд должен быть реализован как [12:0]-разрядный регистр в соответствии с форматом командного слова.

```
reg [12:0] pc, pcN;
```

2.6 Стек данных (dstack)

Стек данных отражает на себе все потоки данных, проходящих через процессор. Его ширина WIDTH будет 32 бита.

```
reg [WIDTH-1:0] st0;       // Top of data stack
reg [WIDTH-1:0] st0N;
reg             dstkW;     // Data stack write`
```

Сам стек определен как массив 32-битных регистров. Количество регистров в массиве 16, но может быть и 32.

2.7 Указатель стека данных (dsp)

Ширина регистра указателя стека данных определяется глубиной (DEPTH=4) стека данных.

```
reg [`DEPTH-1:0] dsp;       // Data stack pointer
reg [`DEPTH-1:0] dspN;
```

2.8 Стек возвратов (rstack)

Стек возвратов строится аналогично стеку данных и имеет те же параметры.

Вершина стека возвратов.

```
wire [`WIDTH-1:0] rstkD; // R stack write value
```

2.9 Указатель стека возвратов (rsp)

```
reg [`DEPTH-1:0] rsp, rspN;
reg rstkW; // R stack write
```

3 Описание команд

3.1 Команда литерал (L)

Первый формат — это литерал. Как видно из формы (1), литерал формируется непосредственно в командном слове. И семантику литерала можно описать следующим образом: это то, что непосредственно представляет некоторое значение. (ГОСТ 28397-89 (ИСО 2382-15-85) Языки программирования. Термины и определения). Интересно, как же определение формы 1 задаёт структуру?

```
  always @* begin // управление стеком данных (АЛУ)
    casez ({insn[15:8]})
      8'b1??_?????:          st0N={{(`WIDTH-15){1'b0}},
insn[14:0]}; // literal
  ...
```

Литерал распознаётся по 1 в старшем разряде командного слова. Остальные 7 разрядов из 8-ми старших разрядов в данном случае роли не играют. Оставшиеся разряды [14:0] распаковываются на вершину стека. Самый старший разряд приобретает значение 0. Поскольку, в данной реализации командное слово имеет 16 бит, остальные 15 бит могут быть заняты значимыми битами литерала. Как интерпретировать эти биты? Да как угодно. Это может быть адрес или код символа, или число со знаком, или число без знака,или ешё что угодно. Просто надо понимать, что это 15 бит, и при этом помнить, что в форте нет типов и точка.

И чтобы команда литерал была выполнена полностью, необходимо сдвинуть указатель стека данных так, чтобы он указывал на новую вершину стека данных. Все дополнительные действия по обработке команд реализованы в устройстве управления.

```
  always @* begin                    // Устройство
управления
    casez ({insn[15:13]})            // стек данных
    3'b1??:   {dstkW, dspI} = {1'b1, 4'b0001};
    ...
    endcase
    dspN = dsp + dspI;
```

Команда литерал не воздействует на стек возвратов. Но после выполнения этой команды, счетчик команд должен быть передвинут на следующее значение, чтобы указывать на следующую команду.

```
  always @* begin                          // Устройство
управления
    ...
    // Управление счетчиком команд
    casez ({reboot, insn[15:13], insn[7], |st0})
      6'b1_???_?_?:   pcN = 0;
      6'b0_000_?_?,
      6'b0_010_?_?,
      6'b0_001_?_0:   pcN = insn[12:0];
      6'b0_011_1_?:   pcN = rst0[13:1];
      default:        pcN = pc_plus_1;
    endcase
  end
```

Здесь будет выполнен случай default, и в итоге счетчик команд будет увеличен на 1.

3.2 Команда безусловного перехода (J)

Следующая форма, на рисунке 1, представляет собой команду безусловного перехода по адресу, который кодируется в битах [12:0]. Старшие три разряда [15:13] представляют код команды. Реализуется это следующим образом:

```
  always @* begin
  casez ({insn[15:8]})
  ...
    8'b000_?????: st0N = st0;  // jump
  ...
```

Содержимое вершины стека не меняется, но для выполнения перехода требуется дополнительная логика.

```
always @* // Устройство управления
...
  casez ({reboot, insn[15:13], insn[7], |st0})
    ...
    6'b0_000_?_?,                    // jump
    6'b0_010_?_?,
    6'b0_001_?_0:   pcN = insn[12:0];
...
```

Здесь в счетчик команд (pcN) записывается адрес перехода, содержащийся в команде jump в битах [12:0]. И следующая команда уже будет считана с этого адреса. Другие биты командного слова никак не влияют на выполнение команды.

3.3 Команда условного перехода (CJ conditional jump)

Команда условного перехода кодируется также битами [15:13], но переход осуществляется при 0 на вершине стека, то есть эту команду можно еще назвать “переход если 0”.

Декодирование команды происходит также в основном цикле

```
  always @* begin  // АЛУ
  casez ({insn[15:8]})
  ...
    8'b001_?????: st0N = st1;  // conditional jump
  ...
```

Дополнительная логика, соответственно, в дополнительном цикле

```
always @* // Устройство управления
...
  casez ({reboot, insn[15:13], insn[7], |st0})
    ...
    6'b0_000_?_?,                    // jump
    6'b0_010_?_?,
    6'b0_001_?_0:   pcN = insn[12:0];
...
```

Здесь в нижней строке добавлено условие 0 из стека st0. Обратите внимание, как это сделано! Таким образом переход будет выполняться, если на вершине стека 0 в самом младшем разряде стека данных. Этот 0 на вершине стека может быть получен в результате любой операции, которую предусмотрит программист.

Таким же образом можно рассмотреть и все остальные процессорные инструкции. Но перейдём к следующему этапу.

4. На пути к toolchain

Можно ли уже на этом этапе заставить процессор выполнить полезную работу? Допустим, необходимо сложить два числа. Для этого надо разместить в стеке два числа и выполнить команду сложение. Результат должен остаться на вершине стека. Число на стек можно положить, используя команду литерал.

```
литерал 1000_0000_0000_0000 h8000
число 1 0000_0000_0000_0001 h0001
```

Поразрядно суммируем и получаем результат

```
L 1000_0000_0000_0001 h8001
```

Это и будет машинная инструкция литерал с числом 1. Таким же образом сформируем второй операнд число 2.

```
L 1000_0000_0000_0010 h8002
```

Теперь сформируем команду сложить (T+N). Для формирования этой команды надо указать несколько полей.

1. Класс команды ALU [15:13] (0110_0000_0000_0000)
2. Код операции T+N [11:8] (0000_0010_0000_0000)
3. Изменение указателя стека данных d-1 3:0

Поразрядно просуммировав эти поля получим

```
T+N 0110_0010_0000_0011 h6203
```

Это и будет машинная инструкция ADD или в нотации J1 T+N. Семантика этой инструкции - получить сумму двух чисел, результат оставить на вершине стека.

Теперь осталось разместить коды инструкции и операндов в памяти процессора:

```
ADDR: CODE BIN            HEX
0000: 1000_0000_0000_0001 h8001
0001: 1000_0000_0000_0010 h8002
0002: 0110_0010_0000_0011 h6203
```

При включении процессор обращается к нулевому адресу ПЗУ. Считывает операнды. Загружает их в стек. И выполняет инструкцию T+N, оставляя на вершине стека результат.

Вот так, от функции мы получили структуру. Это явление наблюдается и в жизни. Когда-то появилась функция, допустим “печень”. Вслед за этим появилась структура “печень“. На протяжении миллионов лет эта структура совершенствовалась, чтобы лучше и лучше выполнять функцию, которая так и осталась “печень”.

Но вернёмся к нашей задаче «помигать светодиодом». Это и есть одна из функций,которая задаёт вектор работ по расширению структуры процессора. То есть мы переходим от самых простых структур с функцией помигать светодиодом, например здесь [2] к более сложным структурам, выполняющих ту же самую функцию, но уже при помощи процессора.

До сих пор мы говорили только о шине, по которой передаются команды. Теперь,когда появилась эта функция помигать светодиодом, мы должны расширить структуру для обеспечения выполнения данной функции. В самом процессоре, конечно же, нет никаких светодиодов. Следовательно, необходимо расширить структуру так, чтобы появилась возможность добавить светодиод. Такое расширение можно реализовать, используя шину данных. Данная структура описывается следующим образом.

```
output wire [`WIDTH-1:0] dout,     // To IO
input  wire [`WIDTH-1:0] mem_din,  // From memory

output wire              io_wr,
output wire [15:0]       mem_addr,
output wire              mem_wr,

input  wire [`WIDTH-1:0] io_din,   // From IO
```

Здесь dout — шина данных для передачи данных в память, либо в систему ввода вывода (ВВ). Из системы ВВ данные можно прочитать на шине io_din. Источником данных на шине dout служит стек данных.

```
assign dout = st1;
```

Данные из системы ВВ по шине io_din попадают в стек при помощи инструкции:

```
always @* // ALU
  begin
    // Compute the new value of st0
    casez ({insn[15:8]})
      ...
      8'b011_?1101: st0N = io_din; // h6D00
      ...
```

Теперь осталось подключить светодиод к шине данных. Это делается за пределами процессора.

4.1 Подключаем светодиод

Там же, за пределами процессора, но внутри верхнего модуля top, создадим регистр gpo

```
reg [127:0] gpo;
```

и цикл управления системой ВВ так, чтобы разряды регистра gpo можно было устанавливать в 1 или в 0, используя шину dout[0] и адрес на шине mem_addr.

```
always @(posedge clk) begin
  casez (mem_addr)
    ...
  endcase
  if (io_wr_) begin
    casez (mem_addr_)
      16'h00??: gpo[mem_addr_[6:0]] <= dout_[0];
      ...
    endcase
  end
end // always @ (posedge clk)
```

И теперь осталось подключить светодиоды, которые имеются на плате, к выходам регистра gpo.

```
assign LEDR[0] = gpo[24]; // 'h18
assign LEDR[1] = gpo[25]; // 'h19
assign LEDR[2] = gpo[26]; // 'h1A
assign LEDR[3] = gpo[27]; // 'h1B
```

После этого можно приступать к написанию программы, которая будет заставлять мигать светодиод, устанавливая соответствующие разряды регистра в 1 и в 0 поочерёдно. Заставим эту структуру работать, выполняя заложенную в неё функцию.

4.2 Программируем мигалку на светодиоде

Примем, что программа будет располагаться начиная с нулевого адреса в ПЗУ. Компилировать будем вручную. Чтобы не сбиться с пути, запишем программу на псевдокоде следующим образом:

```
Начало

расположить по адресу 0000 число 1              ( n )

положить на стек адрес устройства LEDR[0] 'h18 ( n
addr )

отправить число по указанному адресу            ( LEDR
[0]=1 )

записать на вершину стека число 0

положить на стек адрес устройства LEDR[0] 'h18 ( n
addr )

отправить число по указанному адресу            ( LEDR
[0]=0 )

перейти в начало
```

Скомпилируем эту программу.

```
L       1000_0000_0000_0001 h8001 \ литерал 1
L       1000_0000_0001_1000 h8018 \ литерал 'h18
```

Команда литерал автоматически продвигает указатель стека на новую вершину. Теперь надо сформировать команду “отправить число по указанному адресу”. В такой команде надо оформить несколько полей и выполнить пару действий:

- Установить на шине адреса mem_addr значение адреса, взятое с вершины стека данных, передвинуть указатель стека на -1;
- Отправить по указанному адресу значение, взятое с вершины стека данных, передвинуть указатель стека данных на -1.

Или в привычных уже инструкциях, это можно записать так:

```
T         0000_0000_0000_0000         \ Инструкция T
N->[T]    0000_0000_0011_0000         \ Направление
потока
d-1       0000_0000_0000_0011         \ Изменение
указателя стека
alu       0110_0000_0000_0000         \ Форма АЛУ
Итого:    0110_0000_0011_0011 'h6033 \ Код
```

Второе действие сформируем так:

```
N         0000_0001_0000_0000         \ Инструкция N
d-1          0000_0000_0000_0011             \ Изменение
указателя стека
alu       0110_0000_0000_0000         \ Форма АЛУ
Итого:    0110_0001_0000_0011 'h6103 \ Код
```

Скомпилируем команду “перейти в начало” - это просто безусловный переход по адресу 0.

```
ubranch   0000_0000_0000_0000
addr           0000_0000_0000
Итого:    0000_0000_0000_0000 'h0000
```

Разместим всю программу в ПЗУ

```
LABEL  ADDR  CODE BIN              HEX
---------------------------------------
BEGIN: 0000: 1000_0000_0000_0001 'h8001 \ 1
       0001: 1000_0000_0001_1000 'h8018 \ 'h18 LEDR[0]
       0002: 0110_0000_0011_0011 'h6033
       0003: 0110_0001_0000_0011 'h6103 \ LED ON
       0004: 1000_0000_0000_0000 'h8000 \ 1
       0005: 1000_0000_0001_1000 'h8018 \ 'h18 LEDR[0]
       0006: 0110_0000_0011_0011 'h6033
       0007: 0110_0001_0000_0011 'h6103 \ LED OFF
       0008: 0000_0000_0000_0000 'h0000 \ go to BEGIN
```

Конечно, на большой тактовой частоте процессора мигания светодиода не будет видно. И данную программу необходимо дополнить каким-либо счётчиком

для задержки мигания. Но можно просто уменьшить тактовую частоту до приемлемого значения и наблюдать эманацию света из источника LEDR[0], обусловленную исполненным словом.

5 Заключение

Таким образом, вместе с автором J1 был пройден путь от функции “помигать светодиодом” до создания структуры на примере процессора J1, который обеспечивает выполнение заданной функции. Это путь от слова до его материализации в этой вселенной.

Была показана связь между принятыми аппаратными решениями по созданию структуры и последующим программированием действий на данной аппаратной платформе.

Теперь можно вполне осознанно написать программу ассемблер для автоматизированной компиляции программ и другие инструменты, которые обычно входят в toolchain.

.

Список литературы

[1] @nckma. «Процессор Forth J1 в FPGA плате M02mini». В: (). URL: https://habr.com/ru/articles/523348/

[2] @sergebn. «Сага о светодиодах». В: (). URL:(https://fpga-systems.ru/saga-o-svetodiodah)

[3] James Bowman. «J1». В: (). URL: https://github.com/jamesbowman/j1.

[4] James Bowman и Willow Garage. «J1: a small Forth CPU Core for FPGAs». В: (янв. 2010).URL: https://excamera.com/files/j1.pdf.

[5] «Википедия». В: *Гарвардская архитектура* (). URL: (https://ru.wikipedia.org/wiki/Гарвардская_архитектура)

[6] Philip Koopman. Stack Computers: The New Wave. Independently published (February 27, 2023), с. 234. ISBN: 979-8379192419.

Работа с DPI

Куренков Константин

Telegram @kkurenkov

Обсуждение и комментарии :: ссылка

Аннотация

Статья начального уровня для тех, кто совсем не знаком с Direct Programming Interface (DPI). Показаны элементарные приемы и рассмотрены примеры взаимодействия двух языков C - SystemVerilog.

Общие сведения

Direct Programming Interface SystemVerilog (DPI) - это интерфейс взаимодействия между SystemVerilog и другим языком программирования, например языком C или C++. DPI позволяет разработчику легко вызывать функции C из SystemVerilog и экспортировать функции SystemVerilog, чтобы их можно было вызывать из C.

Использование DPI дает несколько преимуществ:

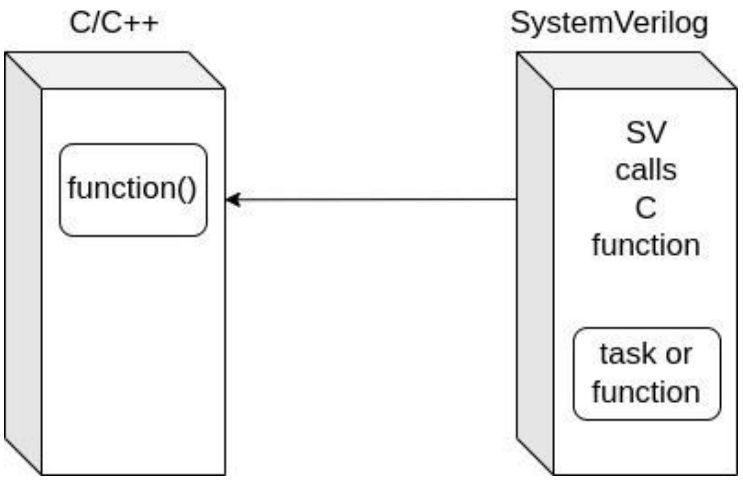
- позволяет пользователю повторно использовать существующий код C;
- не требует знания интерфейсов Verilog Programming Language Interface (PLI) или Verilog Procedural Interface (VPI);
- если математическая модель поведения DUT (Design Under Test — тестируемого модуля) уже написана на Си вам нет необходимости переписывать ее на языке SystemVerilog.

DPI помогает использовать всю мощь и силу C/C++ при разработке тестовых сценариев.

Предположим у нас большой проект, который компилируется 30 минут. Основной смысл состоит в том, что если нам нужно внести изменение в C файлы нам нужно перекомпилировать только C файл, а не весь проект. Это позволяет сэкономить время на разработку.

Вызов DPI функций осуществляется по тем же правилам, что и обычных SystemVerilog функций. Плюсом данного подхода является то, что мы можем сфокусироваться над реализацией нужного функционала на языке Си, вместо того, чтобы думать как перенести этот функционал непосредственно внутрь симуляции.

Импортирование C функций внутрь SV файлов



Функции, реализованные на C, могут быть вызваны из SystemVerilog. Для этого нужно выполнить несколько условий:

- Определить функцию в SystemVerilog
- DPI функции имеют дополнительные ключевое слово за словом import - "DPI-C" или "DPI"
- Определить тип аргументов, входящих в функцию и тип возвращаемого значения

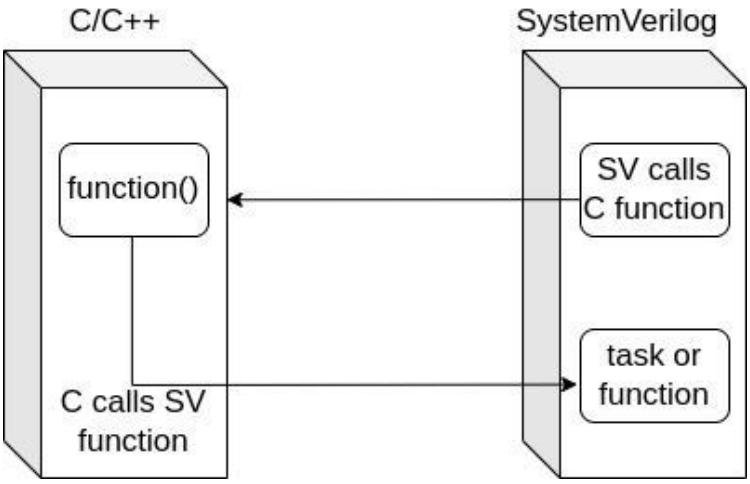
Пример импортирования функции внутрь SystemVerilog файл:

```
import "DPI" function void read(input int address,
output int data);
```

Несколько примеров импорта DPI функций представлено ниже:

```
import "DPI-C" function int add(input int a, input int
b);
import "DPI-C" function void add_output(input int a,
input int b, output int c);
```

Экспорт SV функций внутрь C файлов



Функции, реализованные в SystemVerilog могут быть вызваны в C файлах. Для этого нужно выполнить несколько условий:

- Убедиться, что функция или таск использует ключевое слово *extern* в C файле
- В SV файле функции имеют дополнительные ключевое слово за словом *export* - "DPI-C" или "DPI"

Несколько примеров экспорта DPI функций представлено ниже:

```
export "DPI" function write;
export "DPI-C" function int randomize(int range_a,
range_b);
```

Передача значений по ссылке

Если аргумент для SV функции является возвращаемым (указан как output), то в C функции ему соответствует указатель

```
//SV - passing by reference
function void f1(output int a); // direction of a is
output, thus it is passing a reference
```

```
//C - passing by reference
void f1(int* a); // argument a is passed by reference
```

Основные типы и их преобразование между C - SystemVerilog

DPI определяет трансляцию типов данных между C/C++ и SystemVerilog. Ниже представлена таблица соответствия типов данных.

SystemVerilog Type	C Type	Size
byte	char	1 bytes
shortint	short int	2 bytes
int	int	4 bytes
longint	long long int	8 bytes
real	double	8 bytes
shortreal	float	4 bytes
chandle	void *	1 bytes
string	const char*	1 bytes
bit	unsigned char	1 bytes
logic/reg	unsigned char	1 bytes

Также, в C существуют аналоги сигналов 4 состояний (logic, reg) и с ними также можно взаимодействовать в C функциях. Если подобный функционал вам необходим, то его описание можно найти в книге SystemVerilog for Verification, Third Edition (Chris Spear, Greg Tumbush). Раздел 12.2.4 4-State Values).

Примеры и компиляция проекта

Разберем простой пример. Все примеры хранятся в репозитории [1] и вы можете легко их воспроизвести самостоятельно если у вас:

- Установлен Xcelium.
- Определена переменная окружения CDS_INST_DIR, указывающая на корневую директорию, где установлен Xcelium.
- Установлен make.
- Установлен git.

Все примеры самостоятельно можно преобразовать под запуск в других симуляторах.

1 Простой пример импорта

Для элементарной функции сложения имеется следующая функция на Си.

```
#include <stdio.h>
// функция, принимает на вход два значения типа int и
возвращает сумму этих двух значений
int simple_add(int a, int b) {
  return (a + b);
```

Чтобы использовать данную функцию при симуляции как *shared object* нужно скомпилировать ее, используя стандартный компилятор gcc (в случае C)

Для создания файла .so необходимо выполнить следующую команду в терминале.

```
username:$ gcc simple.c -fPIC -shared -o dpi.so
```

```
// SystemVerilog code
import "DPI-C" function int simple_add(input int a,
input int b);

module top;
  initial begin
    int a, b;
    a = 40;
     b = 2;
     $display("function simple_add --> %0d + %0d = %
0d\n", a, b, simple_add(a, b));
  end
```

Запуск примера осуществляется командой

```
username:$ xrun -sv_lib dpi.so ./tb_top.sv
```

Для автоматизации компиляции .so файла, а также для запуска тестового примера был создан Makefile, который тоже находится в репозитории [1].

Весь пример (компиляцию и запуск) можно выполнить одной командой, находясь в нужной директории

Клонирование репозитория, переход по нужному иерархическому пути и запуск примера выглядит следующим образом:

```
username:$ git clone
username:$ cd ./1
username:$ make
```

Результат выполнения данных действий выдаст в терминал лог с печатью результата:

```
xcelium> run
function simple_add --> 40 + 2 = 42

xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
```

2 Простой пример экспорта

Рассмотрим пример с экспортом.

Запуск примера осуществляется так же - командой make в терминале.

Пример показывает возможность экспортировать SV функцию (sv_display) непосредственно в C, а далее вызвать эту C функцию (c_display) из симуляции.

Отметим необходимость добавления ключевого слова *context* для функции *c_display*. Дело в том, что без него симулятор будет выдавать ошибку.

```
// xmsim: *E,NONCONI: The C identifier "sv_display"
representing an export task/function cannot be
executed from a non-context
```

Сам пример представлен в директории ./2 того же репозитория [1].

```
// SystemVerilog code
// Если вызывать функцию, внутри которой есть вызов SV
(в нашем случае это функция sv_display),
// то обязательно должно быть ключевое слово context,
// иначе будет ошибка
import "DPI" context function void c_display();

export "DPI" function sv_display;

function void sv_display();
  $display("SV: in sv_display\n");
endfunction

module top;
initial
  c_display();
endmodule
```

```
// C code

#include <stdio.h>

extern void sv_display();

int c_display() {
  printf("C: in c_display\n");
  sv_display();
}
```

Результаты симуляции:

```
xcelium> run
C: in c_display
SV: in sv_display

xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
```

Стоит отметить тот факт, что печати C функций, которые выполняются с помощью printf() не отображаются в логе симуляции. В этом можно убедиться, проанализировав лог xrun.log, который находится в той же директории.

Лог, который мы видим в терминальном окне:

```
xcelium> run
C: in c_display
SV: in sv_display

xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
```

Лог, сохранившийся в файле xrun.log:

```
xcelium> run
SV: in sv_display

xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
```

Вывод C части не сохраняется в *.log файле!

Этот факт следует иметь в виду, когда пытаетесь отлаживаться по сохраненным логам.

Рекомендуется завести функцию, аналогичную printf(), но которая будет прокидывать печать не только в терминал, но и сохранять его в лог вместе с временной отметкой.

3 Пример функций печати C функций в логе симулятора.

Пример таких функции представлен также в [репозитории](#) [1] под номером 3.

```
// SystemVerilog code

`include "uvm_macros.svh" // необходим для uvm_error
import uvm_pkg::*;

import "DPI" context function void function_c_display
();
import "DPI" context function void function_c_error ();
import "DPI" context function void function_c_warning
();

export "DPI-C" function throw_display_sv ;
export "DPI-C" function throw_uvm_error_sv ;
export "DPI-C" function throw_uvm_warning_sv;

function void throw_display_sv(string msg);
  $display({"DPI-C INFO: ", msg, $sformatf(" TIME = %
0t", $time())});
endfunction : throw_display_sv

function void throw_uvm_error_sv(string msg);
  `uvm_error("DPI-C", msg)
endfunction : throw_uvm_error_sv

function void throw_uvm_warning_sv(string msg);
  `uvm_warning("DPI-C", msg)
endfunction : throw_uvm_warning_sv

module top;
  initial begin
    function_c_display();
    #100;
    function_c_display();
    #100;
    function_c_error();
    #100;
    function_c_warning();
  end
endmodule
```

```
// C code

#include <stdio.h>
extern void throw_display_sv ();
extern void throw_uvm_error_sv ();
extern void throw_uvm_warning_sv();

void function_c_display() {
throw_display_sv("function_c_display MESSAGE");}
void function_c_error() {
throw_uvm_error_sv("function_c_error MESSAGE");}
void function_c_warning() {
throw_uvm_warning_sv("function_c_warning
MESSAGE");}
```

Для функции *throw_uvm_error_sv* использован UVM макрос *`uvm_error()*.

Теперь лог xrun.log выглядит следующим образом:

```
xcelium> run

.....................

DPI-C INFO: function_c_display MESSAGE TIME = 0
DPI-C INFO: function_c_display MESSAGE TIME = 100
UVM_ERROR ./tb_top.sv(21) @ 200: reporter [DPI-C]
function_c_error MESSAGE
UVM_WARNING ./tb_top.sv(25) @ 300: reporter [DPI-C]
function_c_warning MESSAGE
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
```

4 Пример передачи массивов разной длины из SV в C.

Обычно возникает сложность в передаче массивов из SV в C, заранее не известной размерности.

Предположим, что нужно передавать каждый раз разное количество элементов массива внутрь C функции и выдавать сумму этих элементов.

```
// SystemVerilog code

arr = {4, 5, 6, 7};
ret = summ_array(arr);
arr1 = {8, 9, 10, 11, 12, 13};
ret = summ_array(arr1);

x = 9;
dynArr = new[x];
dynArr = {8, 9, 10, 11, 12, 13, 44, 45, 46};
ret = summ_array(dynArr);
```

Сужествует два варианта решения этой проблемы:

1. Указать максимально возможное количество элементов для суммы и передавать их в C функцию, изначально забивая неиспользуемые элементы нулями.

```
// SystemVerilog code

import "DPI-C" function int summ_array(input int v
[100]);

module top;
  int arr[100];
  int ret;

  initial begin
   foreach (arr[i])
     arr[i] = 0;
     arr = {1, 2, 3, 4};
     ret = summ_array(arr);
     $display("ret = %0d \n", ret);

   foreach (arr[i])
     arr[i] = 0;
     arr = {4, 5, 6, 7, 8, 9, 10};
     ret = summ_array(arr);
     $display("ret = %0d \n", ret);
  end
endmodule
```

```
// C code

int summ_array(const int v[100]){
  int sum;
  for(int i=0; i<100; i++){
    sum = sum + v[i]
  }
  return sum;
}
```

Но данное решение не особо элегантное.

2. Использовать внутренние функции, которые имеются в DPI

```
// SystenVerilog code

import "DPI-C" function int summ_array(input int v[]);

module top;
  int arr[4];
  int dynArr[];
  int ret;
  int x;

  initial begin
    arr = {4, 5, 6, 7};
    ret = summ_array(arr);
    $display("ret = %0d \n", ret);

    x = 6;
    dynArr = new[x];
    dynArr = {8, 9, 10, 11, 12, 13};
    ret = summ_array(dynArr);
    $display("ret = %0d \n", ret);

    x = 9;
    dynArr = new[x];
    dynArr = {8, 9, 10, 11, 12, 13, 44, 45, 46};
    ret = summ_array(dynArr);
    $display("ret = %0d \n", ret);
  end

endmodule
```

```
// C code

#include "svdpi.h" // Добавляем для использования
внутренних типов и функций
#include <stdio.h>

int summ_array(const svOpenArrayHandle v){
int sum;

for(int i=0; i<svLength(v,1); i++){
printf("[%d] = %d \n", i , *((int*)svGetArrElemPtr
(v,i)));
sum = sum + *((int*)svGetArrElemPtr(v,i));
}
return sum;
```

Обратите внимание, что для использования DPI функций и типов данных необходимо к C файлу добавить библиотеку *"svdpi.h"*

Функция *svLength(arg1, arg2)* вычисляет количество элементов в массиве. Здесь *arg1* - указатель на массив v.

arg2 - указание по какой размерности мы идем. (по строке, столбцу и.т.д). У нас одномерный массив, следовательно 1.

Некоторые полезные функции:

- *svLeft()* - возвращает левый индекс массива.
- *svRight()* - возвращает правый индекс массива.
- *svLow()* - возвращает младший индекс массива.
- *svHigh()* - возвращает старший индекс массива.
- *svDimensions()* - показывает размерность массива. (одномерный, двумерный и.т.д)

Существует нюанс, который в нестоящее время не до конца понятен автору, но пары функции *svLeft()* и *svLow()*, *svRight()* и *svHigh()* выдают одинаковые значения. Если вы понимаете в чем разница между функциями - прошу со мной связаться и прояснить этот момент.

Также следует обратить внимание на то, что при компиляции C файла необходимо указать директорию, где располагается файл *svdpi.h*

```
username:$ gcc 4.c -fPIC -shared -o dpi.so -
I${CDS_INST_DIR}/tools.lnx86/include
```

5 Пример с Python функцией

Напрямую подключить Python файлы к SystemVerilog не представляется возможным, но есть обходной путь это сделать используя C прослойку.

Каждый раз при вызове нужной нам функции мы будем инициализировать Python интерпретатор, а по завершению функции выключать его. Возможно такой подход приведет к дополнительным затратам, но подобным исследованием автор не занимался.

Данный способ вполне работоспособен, если возможности переписать эталонную модель на SystemVerilog.

Для этого нам понадобится отыскать дополнительные файлы, которые необходимы для компиляции и симуляции.

- ищем местоположение *libpython3*.so*:

```
username:$ find /usr -name "libpython3.*" 2>/dev/null

/usr/lib/x86_64-linux-gnu/libpython3.10.so.1.0
/usr/lib/x86_64-linux-gnu/libpython3.10.so
/usr/lib/x86_64-linux-gnu/libpython3.10.so.1
```

В данном случае подходит shared library *libpython3.10.so.1*.

- Также при компиляции C файла необходимо добавить файл Python.h, а это значит, что необходимо узнать его месторасположение.

```
username:$ find /usr/ -name "Python.h" 2>/dev/null
/usr/include/python3.10/Python.h
```

Если данные файлы не удаётся найти, следует поставить *python3*. Не обязательно должна быть версия 3.10 (*python 2* такой трюк не сработает).

Теперь рассмотрим процесс компиляции, а потом сам пример. После подставки путей, компиляция кода выглядит следующим образом

```
username:$ gcc -fPIC -shared c_code.c -I/usr/include/
python3.10/ /usr/lib/x86_64-linux-gnu/
libpython3.10.so.1 -o dpi.so
```

py_func(arg1 , arg2) принимает на вход два значения, которые перебрасывается изначально в C функцию, где, в зависимости от второго аргумента выбирается выполнение одной или другой функции.

- При *arg2* = 0 // "ADD" Выполняется суммирование числа само на себя (умножение на два)
- При *arg2* = 1 // "POW_TWO" Выполняется возведение числа в степень 2.

```
presult=PyObject_CallObject(pFunc1,pValue); // 56 line
presult=PyObject_CallObject(pFunc2,pValue); // 71 line
```

Вся магия по запуску и передачи значений Python функции происходит в файле *c_code.c*.

После запуска примера в терминале должен появиться следующий лог:

```
xcelium> run
You passed this Python program Python_Function_pow_two
from C, from system verilog! x * x = 1
Result is 1
inside SV result POW_TWO(1) = 1

You passed this Python program Python_Function_add from
C, from system verilog! x + x = 4
Result is 4
inside SV result ADD(2) = 4

You passed this Python program Python_Function_pow_two
from C, from system verilog! x * x = 9
Result is 9
inside SV result POW_TWO(3) = 9

You passed this Python program Python_Function_add from
C, from system verilog! x + x = 200
Result is 200
inside SV result ADD(100) = 200

xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
```

Заключение

В статье были рассмотрены примеры работы с DPI для обмена данными между SystemVerilog и C, а также между функциями, написанными на языке Python

В последующих публикациях будут разобраны примеры взаимодействия DPI с Open Source AXI и APB агентом.

Список литературы

1. Исходный код к статье https://github.com/kkurenkov/dpi-fpga-systems
2. https://www.amiq.com/consulting/2019/01/30/how-to-call-c-functions-from-systemverilog-using-dpi-c/
3. https://github.com/adki/DPI_Tutorial/tree/main
4. https://www.chipverify.com/systemverilog/systemverilog-dpi
5. https://verificationguide.com/systemverilog/systemverilog-dpi/
6. https://www.doulos.com/knowhow/systemverilog/systemverilog-tutorials/systemverilog-dpi-tutorial/
7. SystemVerilog for Verification, Third Edition (Chris Spear, Greg Tumbush)
8. https://vlsiverify.com/system-verilog/systemverilog-dpi
9. http://www.testbench.in/DP_08_ARRAYS.html
10. https://systemverilog.dev/9.html

Простое вхождение в цифровую схемотехнику с DEEDS

Аверченко А.П., *ОмГТУ*

Telegram @Av_Arte

Обсуждение и комментарии :: ссылка

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

Данная статья демонстрирует основы работы в симуляторе цифровых схем **Deeds-DcS**. Более глубокое понимание Deeds-DcS, а так же описание и использование симулятора конечных автоматов (Deeds-FsM) и эмулятора микрокомпьютера (Deeds-McE) будут рассмотрены в следующих статьях в данном журнале.

Общая концепция состоит в том, чтобы вынести на рабочую область элементы из панели элементов, соединить их проводниками/шинами, сохранить и запустить наглядную симуляцию. Все элементы на панели элементов сгруппированы по функциональной схожести.

Группа «Input Components», представленная на рисунке 1, содержит в себе следующие элементы:

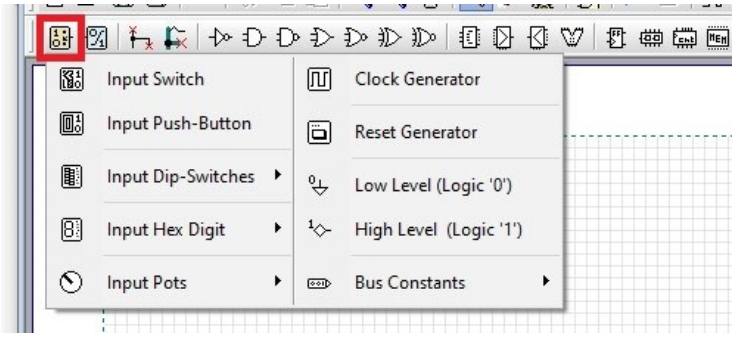


Рисунок 1 — Группа «Input Components»

Imput Swith **-** переключатель, при симулировании может находится в одном из двух состояний: включенном «лог.1» и выключенном «лог.0». Соответственно переключение происходит по клику на данном элементе.

Input Push-Button **-** кнопка, при симуляции имеет состояние выключено «лог.0». Меняет своё состояние на включено «лог.1» в момент нажатия левой кнопки мыши, когда указатель находится на данном элементе, и переходит в выключенное состояние при отпускании левой кнопки мыши.

Input Dip-Switches **-** группа переключателей с различными выходными сигналами и разрядностями. Удобна при задании значений в различных системах счисления.

Input Hex Digit **-** группа компонентов, позволяющая в удобном представлении задавать значения в 16ти-ричной системе счисления с соответствующим отображением установленных значений.

Input Port **-** компоненты, которые позволяют в удобной форме задавать входные оцифрованные сигналы в виде значений с поворотного потенциометра. Данные компоненты имеют только шинные выходы.

Clock Generator **-** генератор тактовых сигналов. Подаёт в схему постоянную тактовую частоту.

Reset Generator **-** генератор начального сброса. Генерирует «лог.0» при запуске симуляции, примерно через 1,5 секунды его выход возвращается к «лог.1». Далее генератор сброса ведёт себя, как кнопка: при нажатии генерируется «лог.0», при отпускании генерируется «лог.1».

Low Level (Logic '0') и High Level (Logic '1') **-** однобитные константы, для подачи в схему логических уровней «лог.0» и «лог.1» соответственно.

Bus Constants **-** шинные константы, которые позволяют подать в схему константные значения различных разрядностей (4,8,16).

Группа «Output Components», представленная на рисунке 2, содержит в себе следующие элементы:

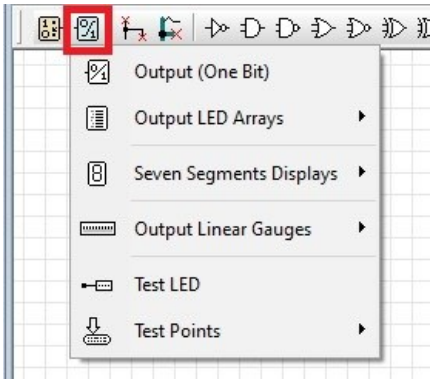


Рисунок 2 — Группа «Output Components»

Output (One Bit) - однобитный компонент. Является выходом элементом схемы. В процессе симуляции отображает значение на подходящем проводнике «лог.0» или «лог.1».

Output LED Arrays - светодиодная матрица на 4, 8, 16 светодиодов, и имеющая как проводниковые, так и шинные входы.

Seven Segments Displays - семисегментные индикаторы, позволяющие отображать числа в 16ти-ричной системе счисления.

Output Linear Gauges - линейный индикатор, позволяющий отображать поданное на него значение в виде светящейся полоски.

Test LED - тестовый светодиод. Своим свечением отображает различные значения на подключённом проводнике. Не является выходным элементом схемы, т.е. при генерации схемы на отладочную плату не принимается, как выходной пин.

Test Points - точки контроля логических значений. Отображают логические уровни в виде цифрового значения (0, 1, X), не являются выходами при переносе схемы в отладочную плату.

Схемы строятся с использованием логических элементов, находящихся на панели элементов, представленных на рисунке 3.

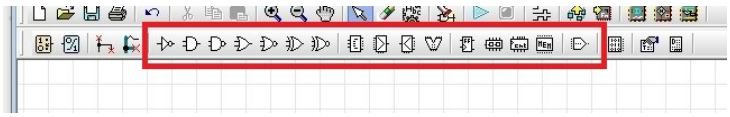


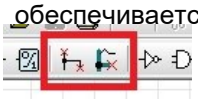
Рисунок 3 - Логические элементы

Предполагается, что все или хотя бы большинство логических элементов уже знакомы читателям. Отдельные пояснения по работе отдельных логических элементов будут представлены в разделах, в которых данные элементы будут использоваться.

При выносе компонента открывается окно, в котором необходимо указать название компонента «Label». Это окно можно оставить незаполненным. В этом случае элементу будет присвоено имя «idXX», где XX - порядковый номер. Рекомендуется задавать имена компонентам в соответствии с логикой работы схемы, что позволит в дальнейшем понять работу разработанной схемы за меньшее количество времени.

Вращение компонентов возможно только при выносе их на рабочую область, пока они являются фантомами. После установки компонента его вращение будет невозможно. Вращение осуществляется нажатием на клавиатуре кнопку «R» или щелчком правой кнопки мыши.

Соединения входов и выходов компонентов обеспечивается при использовании проводников и шин,



которые могут располагаться только по линиям сетки рабочего пространства, т.е. только вертикально или горизонтально. Рисование проводника начинается с выбора данного компонента на панели элементов. Затем необходимо кликнуть левой кнопкой мыши на точке начала линии и протянуть её до точки завершения линии, и нажать левую кнопку мыши. Линия будет нарисована. Для завершения рисования текущей линии нужно нажать правую кнопку мыши. При попытке протянуть линию по диагонали программа сама будет предлагать минимальный маршрут из горизонтальных и вертикальных участков. Однако этот маршрут не всегда является оптимальным и может «заезжать» на другие компоненты, что недопустимо. Для исключения этого необходимо самостоятельно провести линии по вертикальным и горизонтальным направлениям, кликая левой кнопкой мыши в каждой точке перегиба. Завершение рисования линии производится кликом правой кнопкой мыши. Особенно необходимо обращать внимание на прохождение проводников по одним и тем же путям, что приведёт к ошибкам рисования.

Простейшая схема представлена на рисунке 4.

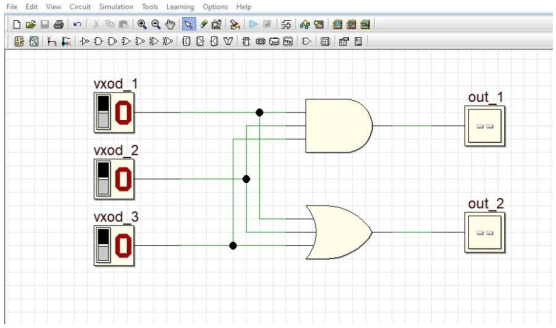


Рисунок 4 - Первая схема в редакторе DEEDS

Запуск симуляции работы схемы производится нажатием на кнопку «Start Animation», находящейся на панели

инструментов и представленной на рисунке 5.

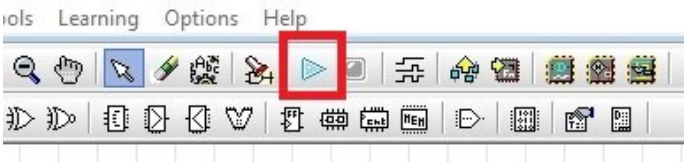


Рисунок 5 - кнопка Start Animation

После запуска симуляции внешний вид панели инструментов и панели элементов меняет свой вид, а схема «включается в работу» и начинает выдавать выходные значения. Как это представлено на рисунке 6.

Во время симуляции пользователь имеет возможность воздействовать практически на любые компоненты группы «Input Components», подавая в схему различные входные значения.

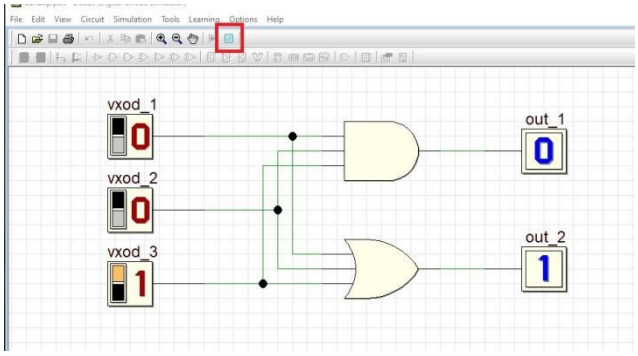


Рисунок 6 - симуляция запущена

Остановка симуляции производится кнопкой «Stop Animation», представленной на рисунке 6.

В следующих статьях будет более детально рассмотрены особенности использования программной среды Deeds, однако уже этого достаточно, чтобы «Просто войти в цифровую схемотехнику с Deeds».

УДК 004.315

Минимизация алгебраических представлений систем булевых функций при синтезе схем модулярных сумматоров и умножителей

Бибило Пётр,
Объединенный институт проблем информатики НАН Беларуси, Республика Беларусь, Минск
e-mail: petr.olibib@yandex.ru

Обсуждение и комментарии :: ссылка

Аннотация

Приведены результаты экспериментов по схемной реализации 2-, 3- и 4-операндных модулярных сумматоров и умножителей в библиотеке проектирования заказных СБИС (сверхбольших интегральных схем) и FPGA (Field-Programmable Gate Array – программируемая пользователем вентильная матрица). Исходные описания проектов модулярных устройств задавались системами не полностью определенных (частичных) булевых функций. Оптимизация алгебраических представлений проводилась в классе дизъюнктивных нормальных форм (ДНФ) – выполнялась совместная минимизация систем булевых функций. Другие виды оптимизации были направлены на минимизацию алгебраических многоуровневых представлений систем булевых функций в классе бинарных диаграмм решений (BDD) и модификаций BDD. Синтезированные схемы оценивались по площади и временной задержке. Установлено, что использование моделей частичных булевых функций позволяет улучшать параметры синтезируемых блоков заказных СБИС и FPGA для небольших значений модуля p, однако лучшие решения для больших значений модуля можно получить, используя алгоритмические VHDL-описания рассматриваемых модулярных устройств. При синтезе схем таких модулярных устройств в составе FPGA и применении системы проектирования Vivado (ф. Xilinx) целесообразно использовать синтезируемые VHDL операции *mod*.

Ключевые слова: модулярные вычисления, многооперандные модулярные сумматоры и умножители, не полностью определенная булева функция, дизъюнктивная нормальная форма, бинарная диаграмма решений, синтез логической схемы, VHDL, FPGA.

Введение

Модулярные вычисления известны достаточно давно [1], в настоящее время они используются не только при программных реализациях алгоритмов модулярной арифметики [2], но и при аппаратной реализации нейросетей [3], алгоритмов цифровой обработки сигналов [4, 5], параллельных алгоритмов обработки изображений [6], криптографических алгоритмов [7, 8] и в других областях. На практике возникает необходимость вычислений как для больших значений, так и для небольших значений модулей, например, в [6] используется система модулей (3, 4, 5), в [9] – система модулей (31, 32, 63).

Аппаратные реализации алгоритмов (не только модулярных) предполагают использование синтезаторов схем, которые позволяют получать логические схемы по спецификациям на специальных языках описания цифровой аппаратуры. В качестве таких языков наибольшее распространение получили VHDL (Very high speed integrated circuits Hardware Description Language – язык описания аппаратуры сверхскоростных интегральных схем) и Verilog [10], а в качестве элементной базы – заказные СБИС (ASIC – Application-Specific Integrated Circuit – специализированная (заказная) интегральная схема) и FPGA [11]. Однако средства для схемной реализации модулярных операций (вычисление значений A *mod* B, нахождение частного и остатка при делении целых чисел) отсутствовали в ранних версиях синтезаторов логических схем, например, они отсутствуют в синтезаторе LeonardoSpectrum (версия 2010a.7) и системе ISE (от англ. Integrated System Environment) Design Suite компании Xilinx (версия 14.7). В настоящее время эти операции реализованы в системе Vivado [10], предназначенной для реализации цифровых систем на основе сложных FPGA [11]. Заметим, что в Vivado в качестве языков для описания проектов цифровых систем кроме VHDL и Verilog используются также языки программирования C, C++ и SystemC [10].

Актуальной является проблема реализации модулярных устройств, прежде всего модулярных сумматоров и умножителей, при их реализации в заказных СБИС и FPGA, в тех случаях, когда синтезаторы не поддерживают модулярные операции (в VHDL это операции *mod* (модуль), *rem* (вычисление остатка от деления), / (вычисление частного). Попытки эффективной реализации модулярных сумматоров и умножителей предпринимались для небольших значений модуля (разрядность операндов не более 7, 8). Для этого в работах [12 - 15] предлагалось использовать системы булевых функций для исходного функционального описания модулярных умножителей, а для уменьшения аппаратной сложности – минимизацию в классе ДНФ систем полностью определенных булевых функций. Экспериментальные исследования схемных реализаций модулярных умножителей для заказных СБИС были проведены в [14], где описаны различные способы алгоритмического описания умножителей, в том числе и с использованием представлений реализуемых систем булевых функций в виде ДНФ, и приведены результаты вычислительных экспериментов. В работах [16, 17] минимизация систем булевых функций в классе ДНФ и многоуровневых представлений была использована для схемной реализации модулярных умножителей: при вычислениях промежуточных сумм требовалось перемножение пары векторов и перемножение двух векторов на константы.

В данной работе, в отличие от [14], предлагается описывать функции модулярных сумматоров и умножителей системами *частичных* (не полностью определенных) булевых функций и осуществлять оптимизацию, выполняя совместную и раздельную минимизацию систем булевых функций в классе ДНФ [18, 19], и в классе BDD-представлений (Binary Decision Diagrams – бинарная диаграмма решений) [20]. Экспериментально установлено, что использование моделей частичных функций и предварительной логической BDD-минимизации позволяет значительно улучшать параметры синтезируемых логических схем по сравнению с использованием функциональных описаний устройств, соответствующих моделям полностью определенных булевых функций.

Схемная реализация VHDL-описаний модулярных сумматоров и умножителей на микросхемах FPGA семейства Kintex-7 [11] (компания Xilinx) осуществлялась в системе автоматизированного проектирования Vivado, в которой реализованы программные средства схемной реализации VHDL-операции *mod* вычисления по модулю (Y<= A *mod* B). В этой системе были проведены сравнения подсхем FPGA, полученных по минимизированным представлениям систем булевых функций, с подсхемами, получаемыми встроенными средствами Vivado, предназначенными для реализации VHDL оператора *mod*. В системе LeonardoSpectrum (версия 2010a.7) VHDL операция *mod* является несинтезируемой, поэтому в экспериментах в этой системе использовались только VHDL модели минимизированных представлений систем булевых функций, а в качестве технологического базиса – библиотека проектирования заказных КМОП СБИС.

1. Модулярные сумматоры и умножители

Умножение по модулю p (основание модулярной вычислительной системы) для двух неотрицательных чисел (операндов) a, b, находящихся в диапазоне $\{0,1,\ldots,p-1\}$ выполняется согласно формуле

$$|a \times b|_p = (a \times b) - \left\lfloor \frac{a \times b}{p} \right\rfloor \times p$$

где через $\lfloor k \rfloor$ обозначена целая часть числа, т. е. ближайшее целое, меньшее либо равное k. В случае если $(a \times b) < p$, то $|a \times b|_p = a \times b$

Сложение по модулю p для двух неотрицательных чисел (операндов) a, b, находящихся в диапазоне $\{0,1,\ldots, p-1\}$ выполняется согласно формулам

$$|a+b|_p = (a+b-p) \text{ , если } (a+b) \geq p \text{ ;}$$
$$|a+b|_p = (a+b) \text{ , если } (a+b) < p \text{ .}$$

Если вместо двух операндов a, b рассматривается три операнда a, b, c, то выражения $a+b$, $a \times b$ заменяются на $a+b+c$, $a \times b \times c$ соответственно. Аналогично при большем трех числе операндов.

2. Генерация матричных форм систем булевых, задающих модулярные сумматоры и умножители

Таблицы истинности, задающие булевы функции модулярных сумматоров и умножителей, были получены выполнением следующих этапов проектирования: сначала были составлены VHDL модели описания функционирования устройств, затем для модели каждого устройства проведено моделирование на всех наборах соответствующего булева пространства. Пример VHDL модели (далее такие модели названы *Alg_model)*

приведен в листинге 1, примеры реакций модели на пяти наборах значений входных сигналов приведены в табл. 1.

Таблица 1. Результаты моделирования VHDL описания умножителя mult_5_3

Входные сигналы	Выходные сигналы
a b c	y
000 000 011	0 0 0
000 000 **101**	- - -
001 001 011	0 1 1
001 100 100	0 0 1
010 **111** 010	- - -

Таблица истинности системы трех частичных булевых функций для модулярного умножителя mult_5_3, получается путем моделирования VHDL описания, заданного в листинге 1, на всех 512 наборах булева пространства девяти входных переменных.

Замечание. В стандарте языка VHDL операции *rem* (нахождение остатка от деления) и *mod* определяются следующим образом.

Для операции L *rem* R должно выполняться соотношение

$$L = (L / R) * R + (L \; rem \; R),$$

где L/R – целая часть частного, (L *rem* R) – результат выполнения операции *rem* (остаток). Остаток имеет ***знак операнда L***.

Для операции L *mod* R должно выполняться соотношение

$$L = N*R + (L \; mod \; R),$$

где N – некоторое целое число, (L *mod* R) – результат выполнения операции *mod*. Результат ***имеет знак операнда R***.

Если L и R – целые положительные числа, то результаты операций L *rem* R, L *mod* R ***одинаковы***.

В пакете IEEE.NUMERIC_STD целые числа со знаком задаются типом SIGNED, целые неотрицательные числа – типом UNSIGNED. Это обстоятельство надо учитывать при составлении функциональных описаний модулярных устройств, операнды которых могут быть в различных диапазонах значений, например, когда некоторые из операндов могут попадать в область отрицательных значений целых чисел. Результаты схемной реализации в системе Vivado могут различаться в зависимости от того, какие целочисленные типы данных SIGNED (со знаком), UNSIGNED (без знака) обрабатываются.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
ENTITY Mult_5_3 IS
    GENERIC (
        n : INTEGER := 3;
        p : INTEGER := 5);
    PORT (
        a, b, c : IN STD_LOGIC_VECTOR(1 TO n);
        y : OUT STD_LOGIC_VECTOR (1 TO n));
END;
ARCHITECTURE beh OF Mult_5_3 IS
    FUNCTION mult
    (SIGNAL a, b, c : STD_LOGIC_VECTOR(1 TO n);
        CONSTANT n, p : INTEGER;
        RETURN STD_LOGIC_VECTOR IS
        VARIABLE y : STD_LOGIC_VECTOR(1 TO n);
        VARIABLE a_int, b_int, c_int, m : INTEGER RANGE
0 TO (2 ** (n)) - 1 := 0;
        VARIABLE r, e : INTEGER RANGE 0 TO 343 := 0;
    BEGIN
        a_int := to_integer(unsigned(a));
        b_int := to_integer(unsigned(b));
        c_int := to_integer(unsigned(c));
        r := (a_int * b_int) * c_int;
        m := r REM p;
        IF ((a_int >= p) OR (b_int >= p) OR (c_int >=
p)) THEN
            y := ('-', '-', '-');
        END IF;
        IF ((r < p) AND (a_int < p) AND (b_int < p) AND
(c_int < p)) THEN
            e := r;
            y := STD_LOGIC_VECTOR(to_unsigned(e, n));
        ELSIF ((r >= p) AND (a_int < p) AND (b_int < p)
AND (c_int < p)) THEN
            e := m;
            y := STD_LOGIC_VECTOR(to_unsigned(e, n));
        END IF;
        RETURN y;
    END mult;
BEGIN
    y <= mult (a, b, c, 3, 5);
END ARCHITECTURE beh;
```

Листинг 1. VHDL описание 3-операндного умножителя по модулю 5

3. VHDL модель системы частичных булевых функций на наборах значений аргументов

В представленном ниже VHDL описании (листинг 2) функций 2-операндного умножителя по модулю 5: $|a \times b|_5 = y$; числа $a=(a_1, a_2, a_3)$, $b=(b_1, b_2, b_3)$ принимают значения из диапазона {0, 1, 2, 3, 4}, вектор $y=(y_1, y_2, y_3)$ задает функции системы { $y_1(a_1,a_2,a_3,b_1,b_2,b_3)$,$y_2(a_1,a_2,a_3,b_1,b_2,b_3)$, $y_3(a_1,a_2,a_3,b_1,b_2,b_3)$ } частичных булевых функций. В листинге 2 вектор $x=(a_1, a_2, a_3, b_1, b_2, b_3)$ – это вектор входных переменных. Неопределенные значения булевых функций y_1,y_2,y_3 обозначены символом «–».

4 Алгоритмы и программы минимизации алгебраических представлений систем булевых функций

Перед выполнением некоторых процедур логической оптимизации проводилось доопределение (замена) неопределенных значений «–» в таблицах истинности частичных булевых функций нулевыми значениями. Далее такое доопределение систем частичных функций до полностью определенных булевых функций далее будет называться «*нулевым*».

4.1. Раздельная и совместная минимизация системы ДНФ булевых функций

Кратчайшей системой ДНФ D_f для системы булевых функций $\boldsymbol{f(x)}=(f_1(\boldsymbol{x}),\ldots,f_m(\boldsymbol{x}))$ называется такая система ДНФ, которая содержит минимальное число *общих* элементарных конъюнкций, на которых заданы ДНФ D_{f_i}, $i=1,\ldots, m$, всех функций f_i системы $\boldsymbol{f(x)}=(f_1(\boldsymbol{x}),\ldots,f_m(\boldsymbol{x}))$.

Задача *совместной минимизации* системы булевых функций: для заданной системы $\boldsymbol{f(x)}=(f_1(\boldsymbol{x}),\ldots,f_m(\boldsymbol{x}))$ булевых функций найти кратчайшую систему ДНФ $D_{\boldsymbol{fi}}$.

Задача *раздельной минимизации* системы булевых функций: найти кратчайшие ДНФ D_f для каждой компонентной функции f_i, $i=1,\ldots, m$, заданной системы $\boldsymbol{f(x)}=(f_1(\boldsymbol{x}),\ldots,f_m(\boldsymbol{x}))$ булевых функций.

В экспериментах совместная минимизация системы частичных булевых функций выполнялась программой Espresso [18], раздельная минимизация – программой Minim [19].

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.all;
entity Mult_5_2 is
port ( x : in std_logic_vector (1 to 6);
y : out std_logic_vector (1 to 3));
end;
architecture BEHAVIOR of Mult_5_2 is
begin
y <= "000" when x = "000000" else
     "000" when x = "000001" else
     "000" when x = "000010" else
     "000" when x = "000011" else
     "000" when x = "000100" else
     "000" when x = "001000" else
     "001" when x = "001001" else
     "010" when x = "001010" else
     "011" when x = "001011" else
     "100" when x = "001100" else
     "000" when x = "010000" else
     "010" when x = "010001" else
     "100" when x = "010010" else
     "001" when x = "010011" else
     "011" when x = "010100" else
     "000" when x = "011000" else
     "011" when x = "011001" else
     "001" when x = "011010" else
     "100" when x = "011011" else
     "010" when x = "011100" else
     "000" when x = "100000" else
     "100" when x = "100001" else
     "011" when x = "100010" else
     "010" when x = "100011" else
     "001" when x = "100100" else
     "---";
end BEHAVIOR;
```

Листинг 2. Не оптимизированное VHDL описание системы частичных булевых функций, задающей функционирование умножителя Mult_5_2

4.2. Минимизация частичных булевых функций в

классе BDD представлений

Графовые BDD представления систем функций строятся на основе разложения Шеннона. Разложением Шеннона полностью определенной либо частичной булевой функции $f=f(\boldsymbol{x})$, $\boldsymbol{x}=(x_1,x_2,\ldots,x_n)$ по переменной x_i называется представление

$$f=f(\boldsymbol{x})=\bar{x}_i f(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n)\vee x_i f(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n) \quad (1)$$

Функции $f_0=f(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n)$, $f_1=f(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n)$ в правой части (1) называются кофакторами (cofactors, англ.) разложения по переменной x_i. Они получаются из функции f подстановкой вместо переменной x_i константы 0 и 1 соответственно. Каждый из кофакторов f_0 и f_1 может быть разложена по одной из переменных из множества $\{x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_n\}$. Процесс разложения кофакторов заканчивается, когда все n переменных будут использованы для разложения, либо когда все кофакторы выродятся до констант 0, 1, «–». Затем выполняется доопределение – замена неопределенных значений листовых определенными 0, 1 с целью минимизации числа кофакторов, т. е. используется *ненулевое* доопределение [20, с. 102]. В результате оптимизации происходит доопределение системы частичных булевых функций до полностью определенных булевых функций.

4.3. Минимизация булевых функций в классе BDDI представлений

Под BDDI (Binary Decision Diagram with Inverse cofactors) понимается ориентированный бесконтурный граф, задающий последовательные разложения Шеннона системы булевых функций $f(x_1,\ldots, x_n)$ по всем ее переменным $x_1,\ldots, x_n$ при заданном порядке (перестановке) переменных, по которым проводятся разложения, при условии нахождения пар *взаимно инверсных* кофакторов [21]. Построение BDDI и соответствующих графу BDDI логических выражений осуществлялось программой BDD_Builder [21] при *нулевом* доопределении исходной системы частичных функций.

4.4. Минимизация булевых функций в классе BBDD и BBDDI представлений

BBDD представления (*Biconditional BDD - BBDD*) строятся на основе разложения

$$f(\boldsymbol{x})=f(x_1,\ldots,x_i,x_j,\ldots,x_n)=$$

$$=(x_i\oplus x_j)f(x_1,\ldots,\bar{x}_j,\ldots,x_j,\ldots,x_n)\vee(x_i\sim x_j)f(x_1,\ldots,x_j,\ldots,x_j,\ldots,x_n) \quad (2)$$

булевой функции $f(\boldsymbol{x})$ по двум переменным x_i, x_j, через « $\oplus$ » обозначена операция исключающее ИЛИ (сумма по модулю 2), через «~» – операция эквиваленция. Разложения (2) подробно изучались в [22] и были названы *biconditional expansion*. В статье рассматриваются совместные BBDD разложения системы $\boldsymbol{f}(\boldsymbol{x})=(f_1(\boldsymbol{x}),\ldots,f_m(\boldsymbol{x}))$ функций, у которых после каждого шага разложения по двум переменным находятся одинаковые остаточные подфункции $s_0=f_q(x_1,\ldots,\bar{x}_j,\ldots,x_j,\ldots,x_n)$, $s_1=f_q(x_1,\ldots,x_j,\ldots,x_j,\ldots,x_n)$ в множестве всех остаточных подфункций компонентных функций $f_q(\boldsymbol{x})$, $q=1,\ldots,m$, исходной системы. Заметим, что после одного шага разложения (2) остаточные подфункции не зависят от переменной x_i. Если при построении BBDD после каждого шага разложений вида (2) всех компонентных функций исходной системы находятся пары взаимно инверсных остаточных подфункций, то в результате строятся BBDDI (BBDD with Inverse subfunctions).

Для многоуровневых разложений BDD, BDDI, BBDD, BBDDI логическая минимизация заключается в нахождении такой последовательности переменных разложения, при которой число подфункций является по возможности наименьшим. Такие разложения строятся по исходным матричным заданиям ДНФ функций и используют сравнение булевых функций (кофакторов), переходя к полиномам Жегалкина [21], что позволяет быстро находить пары взаимно инверсных функций.

5. Организация экспериментов

Для каждого модулярного устройства, функциональное описание которого задавалось матричной формой системы булевых функций, выполнялись следующие три этапа экспериментального исследования.

Этап 1. Логическая минимизация системы частичных булевых функций, заданных таблицей истинности, и получение минимизированной системы полностью определенных булевых функций, заданных логическими уравнениями в том или ином классе алгебраических представлений (ДНФ, BDD, BDDI, BBDDI).

Этап 2. Конвертация минимизированных представлений системы булевых функций в описания на языке VHDL.

Этап 3. Синтез логических схем в базисе проектирования заказных СБИС и базисе FPGA.

Для VHDL моделей устройств, получаемых не логической минимизацией исходных матричных форм частичных функций, выполнялся только этап 3. Примеры таких VHDL моделей заданы в листингах 1 и 2.

В системе FLC-2 [23] выполнялась логическая оптимизация сгенерированных исходных описаний систем частичных функций, задающих функционирование модулярных устройств рассматриваемого класса. Исходные описания для логической оптимизации (таблицы истинности) задавались парой матриц – булевой и троичной матрицей. Булева матрица задавала все наборы булева пространства, троичная – значения частичных булевых функций на соответствующих наборах. Имена и значения параметров сгенерированных функциональных описаний модулярных устройств приведены в табл. 2, где через *Sum_p_k*, *Mult_p_k* обозначено *k*-операндное устройство по модулю *p* – сумматор (*Sum*) и умножитель (*Mult*) соответственно. Например, через *Sum_15_3* обозначен 3 -операндный сумматор по модулю 15.

Параметры *n*, *m* задают число аргументов и функций соответствующей системы частичных булевых функций, через *D* обозначено число наборов, на которых все значения компонентных частичных функций являются неопределенными, т. е. для таких наборов соответствующий *m*-компонентный троичный вектор значений функций состоит только из «–». Множество таких наборов обычно называется в литературе *Don't care*.

Затем осуществлялся перевод минимизированных SF-описаний в описания на языке VHDL. Схемная реализация полученных VHDL-описаний в библиотеке проектирования отечественных КМОП СБИС выполнялась с помощью синтезатора LeonardoSpectrum [24]. Функции базисных логических КМОП элементов используемой библиотеки синтеза приведены в [25]. Для каждого описания схемы модулярного сумматора и умножителя синтез осуществлялся с одними и теми же опциями управления синтезом. Для каждой полученной схемы подсчитывалась площадь схемы (сумма площадей элементов в условных единицах) и временная задержка.

Схемная реализация *алгоритмических* VHDL-описаний модулярных устройств осуществлялась в системе автоматизированного проектирования Vivado (версия 2019.1) [10]. В качестве целевой была выбрана микросхема xc7k70tfbv676-1 семейства Kintex-7 [11], стратегия синтеза – «Vivado Synthesis Defaults», стратегия имплементации – «Vivado Implementation Defaults». При синтезе не использовались блоки DSP (Digital Signal Processor – цифровой сигнальный процессор), поэтому значение опции *-max_dsp* синтеза полагалось равным нулю.

Алгоритмические описания модулярных устройств были двух видов: первый вид – алгоритмические описания, используемые для генерации таблиц истинности модулярных устройств (пример приведен в листинге 1), второй вид состоял из одной операции mod: *y<= A mod p*, где A было равно сумме либо произведению *k* операндов, *k*=2, 3, 4; *p* – значение модуля. Подсхема FPGA, реализующая модулярное устройство, оценивалась по временной задержке и по площади – числу программируемых элементов LUT – настраиваемых элементов, входящих в состав конфигурируемых логических блоков FPGA (Look-Up Table — таблица, реализующая логическую функцию).

Были проведены четыре эксперимента по схемной реализации модулярных устройств.

Эксперимент 1. Синтез схем модулярных сумматоров в библиотеке проектирования заказных СБИС.

Эксперимент 2. Синтез схем модулярных умножителей в библиотеке проектирования заказных СБИС.

Эксперимент 3. Синтез схем модулярных сумматоров в составе FPGA.

Эксперимент 4. Синтез схем модулярных умножителей в составе FPGA.

6. Результаты экспериментов

Результаты экспериментов 1 – 4 представлены в табл. 3 – 6 соответственно. Варианты логической оптимизации и способы исходных VHDL описаний следующие:

Alg_model – способ алгоритмического описания устройства, используемый для генерации системы частичных функций (пример в листинге 1);

BDDI – оптимизация на основе минимизации BDDI представлений систем полностью булевых определенных функций (используется нулевое доопределение системы частичных функций);

Espresso – совместная минимизация системы частичных функций программой Espresso [18];

BDD_partial – оптимизация на основе доопределения и минимизации BDD представлений системы частичных функций [20, с. 201];

BBDDI – оптимизация на основе минимизации BBDDI представлений систем полностью булевых определенных функций (используется нулевое доопределение системы частичных функций);

VHDL_partial – неоптимизированное исходное VHDL описание системы частичных функций, заданных на наборах значений аргументов (пример в листинге 2);

Таблица 2. Имена и параметры функциональных описаний модулярных устройств

Имя схемы	n	m	2^n	D (Don't care)
2-операндные сумматоры				
Sum_5_2	6	3	64	39
Sum_7_2	6	3	64	15
Sum_9_2	8	4	256	175
Sum_13_2	8	4	256	87
Sum_15_2	8	4	256	31
Sum_17_2	10	5	1 024	735
Sum_19_2	10	5	1 024	663
Sum_23_2	10	5	1 024	495
Sum_25_2	10	5	1 024	399
Sum_27_2	10	5	1 024	295
Sum_29_2	10	5	1 024	183
Sum_31_2	10	5	1 024	63
Sum_37_2	12	6	4 096	2 727
Sum_59_2	12	6	4 096	615
Sum_61_2	12	6	4 096	375
3-операндные сумматоры				
Sum_3_3	6	2	64	37
Sum_5_3	9	3	512	387
Sum_7_3	9	3	512	169
Sum_9_3	12	4	4 096	3 367
Sum_13_3	12	4	4 096	1 899
Sum_15_3	12	4	4 096	721
Sum_17_3	15	5	32 768	27 855
4-операндные сумматоры				
Sum_3_4	8	2	256	175
Sum_5_4	12	3	4 096	3 471
Sum_7_4	12	3	4 096	1 695
Sum_9_4	16	4	65 536	58 975
Sum_13_4	16	4	65 536	36 975
Sum_15_4	16	4	65 536	14 911
2-операндные умножители				
Mult_5_2	6	3	64	39
Mult_7_2	6	3	64	15
Mult_9_2	8	4	256	175
Mult_13_2	8	4	256	87
Mult_15_2	8	4	256	31
Mult_17_2	10	5	1 024	735
Mult_29_2	10	5	1 024	183
Mult_33_2	12	6	4 096	3 007
Mult_41_2	12	6	4 096	2 415
Mult_63_2	12	6	4 096	127
3-операндные умножители				
Mult_3_3	6	2	64	37
Mult_5_3	9	3	512	387
Mult_7_3	9	3	512	169
Mult_9_3	12	4	4 096	3 367
Mult_13_3	12	4	4 096	1 899
Mult_15_3	12	4	4 096	721
4-операндные умножители				
Mult_3_4	8	2	256	1 75
Mult_5_4	12	3	4 096	3 471
Mult_7_4	12	3	4 096	1 695
Mult_9_4	16	4	65 536	58 975
Mult_13_4	16	4	65 536	36 975
Mult_15_4	16	4	65 536	14 911

Minim – раздельная минимизация системы частичных булевых функций с помощью программы Minim [19];

VHDL_operator_mod – модель функционального описания модулярного устройства в виде одного VHDL оператора *mod*.

В табл. 3 — 6 используются следующие обозначения:

k_{min} – число элементарных конъюнкций в совместно минимизированной (программой Espresso) системе ДНФ полностью определенных функций;

S_{ASIC} – площадь схемы заказной СБИС в условных единицах;

τ (нс) – задержка схемы;

LUT – число программируемых элементов LUT в FPGA.

Символ # означает, что синтез схемы для данного варианта не выполнялся, символ * означает лучшее решение – схему с наименьшей площадью либо схему с наименьшей задержкой.

Напомним, что синтезатор LeonardoSpectrum, в отличие от Vivado, не имеет средств для схемной реализации VHDL операций *mod* (модуль), *rem* (вычисление остатка от деления) и / (вычисление частного при делении целых чисел). Поэтому при синтезе схем модулярных устройств можно сравнить эффективность использования в качестве исходных описаний оптимизированных представлений систем функций со средствами, имеющимися в системе Vivado (см. столбец «*VHDL_operator_mod*» в табл. 5, 6).

Результаты экспериментов 1 и 2 (см. табл. 3, 4) показывают, что наиболее эффективным способом предварительной оптимизации при синтезе модулярных сумматоров и умножителей в составе заказных СБИС является способ *BDD_partial* и реализующая его программа.

Результаты схемной реализации модулярных сумматоров в составе FPGA (табл. 5) позволяют сделать однозначный вывод о том, что использование моделей частичных функций и их минимизация в классе различных представлений (ДНФ и BDD) не является целесообразным – лучше использовать операцию *mod*, схемные реализации которой являются эффективными как по площади, так и по быстродействию получаемых подсхем FPGA.

Однако для модулярных умножителей ситуация другая. Имеется достаточное число вариантов модулярных умножителей, для задания которых целесообразно использовать их задание в виде системы частичных функций, проводить их минимизацию и использовать минимизированные описания в качестве исходных заданий для синтеза. При большом числе модулярных операций, требуемых при параллельной обработке информации, данный «капельный» эффект может быть значительным. Например, при реализации схемы Mult_17_2 найденное лучшее решение имеет сложность 35 LUT, а реализация в Vivado – 47 LUT. При параллельной обработке 1000 подсхем выигрыш может достигать 12 000 LUT. Можно также заметить (см. табл. 5 и 6), что схемы, реализованные по алгоритмическому описанию *Alg_model*, близки по параметру сложности к схемам, синтезированным по способу *VHDL-operator_mod*.

Если используемые системы синтеза схем по VHDL описаниям *не содержат* средств реализации модулярных операций, то описанный в данной статье подход позволяет быстро провести схемную реализацию модулярных устройств небольшой размерности, используя простые программы генерации функциональных описаний устройств в виде таблиц истинности. Таблицы истинности могут быть использованы для минимизации различных представлений логических функций устройств и получения минимизированных описаний. Результаты экспериментов показывают сложность получаемых схем и ограничение по числу входных портов модулярных устройств – не более 16 входных битовых сигналов. Однако, если стоит задача получения схем с лучшими параметрами, то для составления функциональных описаний модулярных устройств требуется использовать уже известные нетривиальные теоретические подходы [26 – 29 и др.], особенно, если требуются описания сумматоров и умножителей со значениями модулей $p=2^n-1$, $p=2^n$, $p=2^n+1$, для которых развиваются специальные методы проектирования, описанные, например, в [26, 27]. Примеры функциональных описаний для 2-операндных модулярных сумматоров можно найти в [30].

Заключение

Результаты проведенных экспериментов показывают эффективность применения различных программ логической минимизации при реализации модулярных сумматоров и умножителей в составе заказных КМОП СБИС в том случае, когда синтезаторы не имеют встроенных средств схемной реализации модулярных операций. Эксперименты показали преимущество использования моделей частичных функций и минимизации систем частичных функций классе BDD представлений по сравнению с моделями полностью определенных булевых функций и другими программами

минимизации представлений таких систем функций.

При синтезе подсхем модулярных сумматоров в FPGA и применении системы Vivado логическая минимизация таблиц истинности не дает преимуществ в схемной реализации, поэтому для модулярных сумматоров вместо моделей систем булевых функций целесообразно использовать соответствующие типы данных языка VHDLи синтезируемые операции *mod* для этих типов. Для модулярных умножителей имеется достаточное число случаев, когда минимизация табличных представлений систем частичных функций (при небольших значениях модулей) может иметь преимущество по сравнению со средствами проектирования, реализованными в Vivado.

Литература

1. Акушский И. Я., Юдицкий Д. И. Машинная арифметика в остаточных классах. – М.: Советское радио, 1968. – 440 с.

2. Чернявский А. Ф., Данилевич В. В., Коляда А. А., Селянинов М. Ю. Высокоскоростные методы и системы цифровой обработки информации. – Мн.: Белгосуниверситет, 1996. – 375 с.

3. Червяков Н. И., Сахнюк П. А., Шапошников А. В., Рядов С. А. Модулярные параллельные вычислительные структуры нейропроцессорных систем. – М.: Физматлит, 2003. – 288 с.

4. Стемпковский А. Л., Корнилов А. И., Семенов М. Ю. Особенности реализации устройств с цифровой обработкой сигналов в интегральном исполнении с применением модулярной арифметики // Информационные технологии. – 2004. – № 2. – С. 2 – 9.

5. Соловьев Р. А. Микроэлектронные устройства цифровой обработки сигналов на базе модулярных вычислительных структур. Автореферат дисс. на соискание ученой степени доктора техн. наук. М.; 2018. – 39 с.

6. Salamat S., Shubhi S., Khaleghi B., Rosing T. Residue-Net: Multiplication-free Neural Network by In-situ No-loss Migration to Residue Number Systems. 26th Asia and South Pacific Design Automation Conference (ASP-DAC), 2021.

7. Sarada V. An Enhanced Residue Modular Multiplier for Cryptography // International Journal of Science and Research (IJSR). – 2015. – V. 4. – № 8. – P. 2029 – 2034.

8. Коледа А. А., Чернявский А. Ф. Умножение по большим модулям с использованием минимально избыточной модулярной схемы Монтгомери // Информатика. – 2010 – № 3. – С. 31 – 48.

9. Samimi N., Kamal M., Afzalli-Kusha A., Pedram M. Res-DNN: A residue number system-based DNN accelerator unit. // IEEE Trans. Circuits and Systems. – 2020. – V. 67. – № 2. – P. 658 – 671.

10. Тарасов И. Е. ПЛИС Xilinx. Языки описания аппаратуры VHDL и Verilog, САПР, приемы проектирования. // М.: Горячая линия-Телеком. – 2020. – 538 с.

11. Соловьев, В. В. Архитектуры ПЛИС фирмы Xilinx: FPGA и CPLD 7-й серии. – М.: Горячая линия – Телеком, 2016. – 392 с.

12. Амербаев В. М., Соловьев Р. А., Тельпухов Д. В. Реализация библиотеки модульных арифметических операций на основе алгоритмов минимизации логических функций // Известия ЮФУ. Технические науки, Таганрог. – 2013. – №. 7. – С. 221 – 225.

13. Амербаев В. М., Соловьев Р. А., Тельпухов Д. В., Щелоков А. Н. Исследование эффективности модулярных вычислительных структур при проектировании аппаратных однотактных умножителей // Известия ЮФУ. Технические науки, Таганрог. – 2014. – № 7 (156). – С. 248 – 254.

14. Соловьев Р. А., Тельпухов Д. В., Балака Е. С., Рухлов В. С., Михмель А. С. Особенности проектирования модулярных умножителей с помощью современных САПР // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). – 2016. – № 1. – С. 249 – 254.

15. Сравнительное исследование и анализ методов аппаратной реализации сумматоров по модулю / Е. С. Балака [и др.]. // Universum: Технические науки: электрон. научн. журн. – 2016. – № 1 (23). http://7universum.com/ru/tech/archive/item/2887

16. Gorodecky D., Sousa L. Two-Operand Modular Multiplication to Small Bit Ranges // Advanced Boolean Techniques. Selected Papers from the 15th International Workshop on Boolean Problems. Ed. R. Drechsler, S. Huhn. –Springer, 2023. – P. 111 – 122.

17. Gorodecky D., Sousa L. Modular Multiplication Based on Boolean Representations. https://www.researchgate.net/publication/363855664

18. Brayton K. R., Hactel G. D., McMullen C. T., Sangiovanni-Vincentelli A. L. Logic minimization algorithm for VLSI synthesis. Boston, e. a.: Kluwer Academic Publishers, 1984. – 193 p.

19. Торопов Н. Р. Минимизация систем булевых функций в классе ДНФ // Логическое проектирование. – Минск: Инт техн. кибернетики НАН Беларуси, 1999. – Вып.4. – С. 4–19.

20. Бибило П. Н. Применение диаграмм двоичного выбора при синтезе логических схем – Минск: Беларус. навука, 2014. – 231 с.

21. Бибило П. Н., Ланкевич Ю. Ю. Использование полиномов Жегалкина при минимизации многоуровневых представлений систем булевых функций на основе разложения Шеннона // Программная инженерия. – 2017. – № 8. – С. 369 –384.

22. Amaru L. G. New Data Structures and Algorithms for Logic Synthesis and Verification. − Springer, 2017. − 156 p.

23. Бибило П. Н., Романов В. И. Система логической оптимизации функционально-структурных описаний цифровых устройств на основе продукционно-фреймовой модели представления знаний // Проблемы разработки перспективных микро- и наноэлектронных систем. – 2020. Сб. трудов / под общ. ред. акад. РАН А.Л. Стемпковского. – М.: ИППМ РАН, 2020. – № 4. – С. 9 – 16.

24. Бибило П. Н. Системы проектирования интегральных схем на основе языка VHDL. StateCAD, ModelSim, LeonardoSpectrum. – М.: СОЛОН-Пресс, 2005. – 384 с.

25. Бибило П. Н., Кириенко Н. А. Оценка энергопотребления логических КМОП-схем по их переключательной активности // Микроэлектроника. – 2012. – № 1. – С. 65 – 77.

26. Корнилов А. И., Калашников В. С., Ласточкин О. В., Семенов М. Ю. Особенности построения умножителей по модулю (2^n – 1) // Изв. вузов. Электроника. – 2006. – № 1. – С. 55 – 59.

27. Корнилов А. И., Исаева Т. Ю., Семенов М. Ю. Методы логического синтеза сумматоров с ускоренным переносом по модулю (2^n – 1) на основе BDD-технологии // Изв. вузов. Электроника. – 2004. – № 3. – С. 54 – 60.

28. Корнилов А. И., Семенов М. Ю., Калашников В. С. Методы аппаратной оптимизации сумматоров для двух операндов в системе остаточных классов // Изв. вузов. Электроника. – 2004. – № 1. – С. 75 – 82.

29. Стемпковский А. Л., Корнилов А. И., Семенов М .Ю., Ласточкин О. В., Калашников В. С. Построение систем повышенной надежности на основе аппарата модулярной арифметики с применением современных методов и средств проектирования // Проблемы разработки перспективных микроэлектронных систем – 2006. Сборник научных трудов / под общ. ред. А. Л. Стемпковского. – М.: ИППМ РАН, 2006. – С. 253 – 258.

30. Генераторы Verilog. Модулярные сумматоры. http://vscripts.ru/w/Main.

Таблица 3. Схемная реализация модулярных сумматоров в библиотеке проектирования заказных СБИС

Имя схемы	*BDDI*		*ESPRESSO*			*BDD_partial*		*BBDDI*		*VHDL_partial*		*Minim*	
	S_{ASIC}	τ	k_{mim}	S_{ASIC}	τ	S_{ASIC}	τ	S_{ASIC}	τ	S_{ASIC}	τ	S_{ASIC}	τ
2-операндные сумматоры													
Sum_5_2	11 779	3.36	18	12 912	***3.12**	***8 348**	3.62	12 667	4.41	13 392	5.71	12 912	3.12
Sum_7_2	16 946	3.51	36	***13 961**	***2.16**	15 172	4.51	23 425	5.12	***13 961**	***2.16**	19 508	3.87
Sum_9_2	30 260	4.90	55	34 473	***4.62**	***23 196**	5.20	35 723	5.04	41 404	5.84	40 076	4.61
Sum_13_2	41 420	4.84	113	54 723	4.78	***28 971**	4.59	79 108	6.47	37 525	***4.15**	61 246	5.46
Sum_15_2	45 181	5.74	126	51 492	3.94	***18 922**	***3.73**	61 006	6.44	37 453	4.24	63 355	4.56
Sum_17_2	49 500	6.80	150	98 738	6.07	***42 508**	***5.65**	70 788	6.00	108 280	8.02	89 431	5.06
Sum_19_2	55 236	5.46	198	117 652	6.52	***31 153**	***5.35**	96 288	7.28	136 386	8.50	96 484	6.30
Sum_23_2	61 380	7.09	276	131 136	8.28	***32 386**	***5.42**	121 075	7.95	109 552	7.72	128 413	6.09
Sum_25_2	60.454	6.56	315	182 656	7.74	***43 083**	***5.44**	169 805	8.73	75 313	5.49	178 415	6.56
Sum_27_2	78 259	6.13	361	191 160	8.04	***47 487**	***4.78**	177 098	8.81	83 633	5.70	191 656	7.61
Sum_29_2	77 501	6.88	390	180 714	7.56	***44 021**	***5.08**	132 140	7.30	99 106	6.31	209 456	7.35
Sum_31_2	68 991	8.17	356	156 145	6.48	***27 950**	***4.09**	95 078	7.62	76 998	5.69	182 991	6.91
Sum_37_2	83 242	7.23	563	340 068	9.22	***52 430**	***6.78**	169 660	9.77	563 329	12.08	343 421	7.97
Sum_59_2	102 443	9.00	1105	614 012	10.09	***54 634**	***6.60**	197 080	11.12	242 434	7.52	705 714	10.76
Sum_61_2	111 734	8.11	1108	593 349	9.96	***60 627**	7.20	181 266	10.38	226 548	***7.02**	676 675	9.70
3-операндные сумматоры													
Sum_3_3	9 559	3.46	18	***5 764**	***2.76**	8 169	3.80	13 292	3.73	8 420	2.83	11 567	3.18
Sum_5_3	***24 485**	5.73	91	43 390	4.47	26 617	6.49	55 454	5.33	36 940	***4.41**	50 934	5.45
Sum_7_3	39 713	6.67	232	129 216	6.90	***36 013**	***5.98**	95 228	7.61	98 972	6.42	119 697	6.45
Sum_9_3	57 630	***7.06**	457	254 275	8.74	***83 527**	7.88	129 847	9.82	160 905	11.80	260 731	8.55
Sum_13_3	111 198	9.32	1236	771 541	11.10	***78 884**	***7.16**	498 071	10.82	435 486	9.23	730 372	11.90
Sum_15_3	126 097	8.86	1677	1 184 707	11.98	***82 545**	8.69	406 168	11.11	559 027	***8.36**	1 190 521	13.55
Sum_17_3	154 276	10.21	2277	1 541 001	12.76	***105 618**	***10.09**	586 603	12.25	1 367 569	14.84	1 825 173	14.87
4-операндные сумматоры													
Sum_3_4	13 074	4.42	54	***7 488**	***2.30**	12 410	5.57	23 760	5.31	11 417	2.66	***7 488**	***2.30**
Sum_5_4	***36 940**	7.99	449	119 992	7.45	38 268	8.70	90 435	7.45	67 217	***6.59**	131 883	7.52
Sum_7_4	101 606	***6.27**	1 528	806 210	12.94	***58 177**	8.41	134 205	10.37	489 506	10.37	770 185	14.52
Sum_9_4	***89 620**	10.98	4 005	2 274 107	16.06	107 839	10.65	208 659	12.36	226 838	***10.39**	2 444 246	13.36
Sum_13_4	***163 533**	12.56	14 669	#	#	173 772	***11.38**	1 077 682	15.28	2 316 515	14.22	#	#
Sum_15_4	182 527	13.67	22 573	#	#	***144 734**	***11.27**	1 258 072	13.92	#	#	#	#
Число лучших решений	**4**	**2**	**-**	**3**	**5**	**21**	**15**	**0**	**0**	**1**	**7**	**1**	**1**

Таблица 4. Схемная реализация модулярных умножителей в библиотеке проектирования заказных СБИС

Имя схемы	*BDDI*		*ESPRESSO*			*BDD_partial*		*BBDDI*		*VHDL_partial*		*Minim*	
	S_{ASIC}	τ	k_{mim}	S_{ASIC}	τ	S_{ASIC}	τ	S_{ASIC}	τ	S_{ASIC}	τ	S_{ASIC}	τ
2-операндные умножители													
Mult_5_2	10 474	3.04	15	6 819	3.28	***6 601**	***2.40**	12 349	2.88	9 012	3.91	11 311	3.14
Mult_7_2	13 883	2.95	18	11 964	2.61	12 159	2.53	13 554	3.16	***10 608**	***2.10**	74 136	6.69
Mult_9_2	32 615	4.78	49	***29 211**	***4.28**	32 085	5.20	45 215	6.10	35 266	4.85	29 853	3.57
Mult_13_2	69 973	6.11	96	65 799	5.58	65 135	***5.26**	86 334	7.71	***56 570**	5.39	62 596	4.68
Mult_15_2	63 143	5.69	112	74 917	6.27	46 727	5.57	***40 572**	***5.49**	71 084	5.50	73 689	5.85
Mult_17_2	119 362	7.26	180	115 484	7.54	113 006	***6.75**	133 714	7.49	***106 014**	7.59	118 737	7.19
Mult_29_2	275 702	9.19	525	383 647	10.83	***272 817**	8.45	369 530	10.37	421 374	***7.04**	366 249	9.71
Mult_33_2	***302 057**	9.71	645	520 491	10.11	309 361	***9.63**	347 277	10.10	527 165	11.78	459 206	11.62
Mult_41_2	***659 210**	***10.43**	1163	1 000 784	11.54	755 744	11.40	829 545	13.35	804 084	11.26	814758	12.54
Mult_63_2	375 395	10.74	2567	2 475 243	15.09	404 556	10.99	***274 603**	11.77	2 066 787	***9.84**	2 413 423	17.58
3-операндные умножители													
Mult_3_3	5 586	2.66	8	***2 466**	***1.34**	4 871	2.61	5 630	2.49	5 669	2.43	5 669	2.43
Mult_5_3	30 076	4.46	54	23 648	4.79	***23 235**	***4.11**	41 515	4.94	32 682	5.52	33 062	5.13
Mult_7_3	28 837	4.36	108	28 804	***4.34**	37 804	5.33	***22 761**	4.92	62 200	5.47	74 136	6.69
Mult_9_3	102 181	8.99	343	179 079	9.18	***91 557**	8.28	118 698	***8.10**	102 069	10.39	176 428	8.78
Mult_13_3	541 187	10.16	1 043	691 351	11.67	***317 932**	10.90	723 330	13.98	736 153	***9.26**	649 986	12.28
Mult_15_3	128 552	9.00	1 607	1 230 903	11.89	119 775	8.29	***81 909**	***7.86**	1 159 290	11.18	1 190 521	13.55
4-операндные умножители													
Mult_3_4	7 857	3.79	16	***2 991**	***1.41**	7 360	3.78	6 852	2.19	7 645	2.32	***2 991**	***1.41**
Mult_5_4	43 836	***4.50**	197	***39 607**	4.79	40 684	5.32	76 033	6.53	54 243	5.90	42 810	4.95
Mult_7_4	***37 849**	6.50	648	59 170	***6.29**	89 693	9.69	***37 849**	6.50	121 594	8.93	357 656	10.60
Mult_9_4	***134 283**	9.21	2 377	1 348 714	16.66	148 880	***8.64**	306 191	12.63	308 356	12.01	1 407 153	14.47
Mult_13_4	686 106	15.23	11 505	#	#	***630 501**	15.27	1 777 766	17.47	6 641 104	16.15	#	#
Mult_15_4	187 533	12.72	21 733	#	#	225 382	12.94	***139 243**	***11.53**	#	#	#	#
Число лучших решений	**4**	**2**	**-**	**4**	**5**	**6**	**6**	**6**	**4**	**3**	**4**	**1**	**1**

Таблица 5. Схемная реализация модулярных сумматоров в библиотеке проектирования FPGA

Имя схемы	*Alg_model*		*VHDL_partial*		*Espresso*		*BDD_partial*		*BDDI*		*BBDDI*		*VHDL_operator_mod*	
	LUT	τ	LUT	τ	LUT	τ	LUT	τ	LUT	τ	LUT	τ	LUT	τ
2-операндные сумматоры														
Sum_5_2	***3**	***3.295**	***3**	5.972	***3**	6.057	***3**	5.995	***3**	5.916	***3**	6.071	***3**	5.974
Sum_7_2	***3**	5.974	***3**	***5.915**	***3**	5.974	***3**	5.974	***3**	5.974	***3**	5.974	***3**	5.974
Sum_9_2	15	6.686	14	6.618	15	6.645	15	6.830	15	6.532	15	6.778	***7**	***6.525**
Sum_13_2	10	7.005	20	6.739	16	6.612	15	6.673	16	6.700	16	***6.360**	***7**	6.638
Sum_15_2	11	7.303	20	6.823	16	6.651	8	6.553	17	6.509	16	***6.370**	***7**	6.566
Sum_17_2	11	7.480	39	7.329	66	7.817	34	7.560	34	7.814	41	7.624	***8**	***7.113**
Sum_19_2	14	7.255	50	7.674	48	7.353	26	7.659	47	7.670	44	7.660	***8**	***7.113**
Sum_23_2	15	7.457	45	7.478	48	7.550	19	7.279	41	7.441	40	7.627	***8**	***7.218**
Sum_25_2	13	7.633	54	7.635	68	7.888	28	7.213	49	7.589	65	7.755	***8**	***7.203**
Sum_27_2	13	7.341	53	7.555	85	7.994	24	7.372	64	7.965	80	7.728	***8**	***7.203**
Sum_29_2	13	***7.076**	55	7.555	79	8.178	28	7.373	55	7.496	80	7.680	***8**	7.209
Sum_31_2	13	7.415	47	7.462	48	7.613	21	7.221	47	7.450	71	7.874	***10**	***6.629**
Sum_37_2	16	7.951	84	8.431	103	8.998	47	7.797	78	8.258	76	8.751	***12**	***7.324**
Sum_59_2	16	7.951	97	8.515	99	8.461	46	7.935	105	9.579	121	8.766	***12**	***7.324**
Sum_61_2	16	7.885	93	8.891	88	8.435	47	7.815	107	8.692	122	8.794	***12**	***7.324**
3-операндные сумматоры														
Sum_3_3	***2**	5.877	***2**	5.787	***2**	5.791	***2**	5.877	***2**	5.877	***2**	***5.780**	***2**	5.877
Sum_5_3	11	7.310	21	6.875	27	7.483	12	***6.493**	20	7.164	18	6.878	***8**	6.583
Sum_7_3	11	7.360	27	6.999	27	7.482	18	6.889	27	6.979	27	7.022	***8**	***6.591**
Sum_9_3	27	8.129	94	8.571	274	10.784	60	8.721	109	8.831	102	8.560	***12**	***7.036**
Sum_13_3	20	8.070	146	8.797	285	10.606	66	8.271	176	9.942	161	9.187	***12**	***7.300**
Sum_15_3	23	8.031	173	9.000	246	9.623	53	7.803	200	9.268	188	8.924	***12**	***7.259**
4-операндные сумматоры														
Sum_3_4	8	***6.237**	8	6.483	7	6.475	***4**	6.415	***4**	6.479	***4**	6.515	***4**	6.584
Sum_5_4	14	8.120	44	7.917	201	9.343	45	8.390	48	8.066	48	8.003	***10**	***7.557**
Sum_7_4	14	7.937	53	7.885	182	8.927	60	7.728	69	8.280	111	8.982	***11**	***7.470**
Sum_9_4	31	9.286	199	10.496	3189	21.809	166	10.126	177	10.387	198	10.400	***18**	***8.737**
Sum_13_4	28	8.590	#	#	15 694	40.447	245	11.729	293	11.514	346	11.795	***18**	***7.982**
Sum_15_4	29	8.942	#	#	28 483	58.453	240	10.293	368	11.142	868	13.614	***18**	***8.291**
Число лучших решений	**3**	**3**	**3**	**1**	**3**	**0**	**4**	**1**	**4**	**0**	**4**	**3**	**27**	**19**

Таблица 6. Схемная реализация модулярных умножителей в библиотеке проектирования FPGA

Имя схемы	*Alg_model*		*VHDL_partial*		*Espresso*		*BDD_partial*		*BDDI*		*BBDDI*		*VHDL_operator_mod*	
	LUT	τ	LUT	τ	LUT	τ	LUT	τ	LUT	τ	LUT	τ	LUT	τ
2-операндные умножители														
Mult_5_2	***3**	5.974	***3**	5.978	***3**	5.978	***3**	5.974	***3**	5.978	***3**	6.071	***3**	***5.971**
Mult_7_2	***3**	5.974	***3**	5.915	***3**	***5.907**	***3**	5.978	***3**	5.974	***3**	5.974	***3**	5.971
Mult_9_2	39	9.744	15	6.852	15	6.752	***13**	***6.475**	15	6.823	15	6.559	32	8.422
Mult_13_2	39	9.927	20	6.739	***16**	6.655	***16**	***6.343**	***16**	6.475	***16**	6.349	32	8.545
Mult_15_2	39	9.681	20	5.086	***16**	***6.347**	***16**	6.376	***16**	6.461	***16**	6.376	32	8.474
Mult_17_2	57	13.153	36	7.388	46	7.770	***35**	***7.567**	46	7.583	52	7.632	47	12.060
Mult_29_2	64	13.232	99	7.979	85	***7.734**	84	7.984	85	7.969	85	7.861	***55**	12.824
Mult_33_2	93	16.409	330	10.198	132	10.103	128	***8.558**	132	8.960	184	9.602	***89**	14.952
Mult_41_2	88	15.624	295	9.993	383	10.471	254	***9.722**	383	10.599	258	10.136	***79**	15.406
Mult_63_2	70	16.184	214	9.554	408	9.799	171	***9.151**	408	10.101	164	9.572	***68**	15.023
3-операндные умножители														
Mult_3_3	***2**	5.877	***2**	5.787	***2**	***5.780**	***2**	5.877	***2**	5.877	***2**	5.877	***2**	5.877
Mult_5_3	35	10.415	15	6.900	***13**	7.060	***13**	6.960	15	7.285	***13**	***6.665**	28	9.913
Mult_7_3	38	10.367	19	6.983	12	6.967	19	7.181	12	***6.786**	***11**	7.414	33	10.063
Mult_9_3	82	15.293	91	***8.371**	213	9.218	93	8.505	99	8.542	108	8.557	***74**	15.133
Mult_13_3	85	14.600	186	***8.493**	272	9.436	183	9.203	237	9.117	184	8.705	***74**	14.717
Mult_15_3	84	15.847	125	8.652	85	8.526	***67**	8.328	67	***8.268**	***67**	8.287	74	14.498
4-операндные умножители														
Mult_3_4	24	9.684	4	6.376	***3**	6.453	***3**	6.406	***3**	6.406	***3**	***6.010**	19	9.076
Mult_5_4	71	14.182	31	7.684	44	8.258	31	7.877	***22**	***7.388**	30	7.439	59	13.509
Mult_7_4	73	14.177	44	8.206	24	7.820	51	8.155	***21**	***7.779**	***21**	***7.779**	59	13.509
Mult_9_4	147	20.377	212	10.972	544	13.102	182	10.292	182	10.406	169	***10.084**	***137**	19.689
Mult_13_4	149	21.469	#	#	11 518	34.970	475	14.034	366	***11.272**	352	11.571	***137**	20.650
Mult_15_4	149	21.062	#	#	30 770	54.462	211	10.786	***111**	***10.204**	122	10.306	137	20.612
Число лучших решений	**3**	**0**	**3**	**2**	**7**	**4**	**10**	**6**	**9**	**6**	**10**	**4**	**11**	**1**

Стили и способы описания конечных автоматов на языках Verilog и SystemVerilog

Соловьев В.В., *д.т.н., профессор*

e-mail: valsol@mail.ru

Обсуждение и комментарии :: ссылка

Аннотация

Рассматриваются полезные стили и способы описания конечных автоматов на языке Verilog, которые могут также использоваться в языке SystemVerilog. Для каждого способа отмечаются особенности реализации и поведения конечного автомата, а также указывается на возможность практического применения. Показано, что способ описания конечного автомата в некоторых случаях позволяет увеличить быстродействие конечного автомата при его реализации в FPGA более чем в 3 раза.

1. Введение

В цифровых устройствах и системах конечные автоматы чаще всего выполняют роль контроллеров или устройств управления. Поскольку большинство цифровых устройств (а цифровые системы все) имеют устройства управления, конечные автоматы очень широко используются в цифровой аппаратуре. Более того, многие цифровые системы и устройства как раз и отличаются своими устройствами управления. Поэтому при разработке каждого нового проекта остро стоит вопрос проектирования конечных автоматов. Для этого конечный автомат должен быть описан на одном из языков описания аппаратуры (hardware description language – HDL). В литературе можно встретить большое разнообразие стилей описания конечных автоматов на языках HDL, например, с 3 процессами, с 2 процессами или с 1 процессом [1, 2].

Введем различие между стилем и способом описания конечных автоматов. *Стиль описания* (description style) – это общая структура описания конечного автомата, например, с 3 процессами, с 2 процессами или с 1 процессом. *Способ описания* (description method) – это конкретный прием или метод, с помощью которого описывается данный процесс. Каждый стиль и способ описания имеет свои положительные и отрицательные стороны, поэтому нельзя выделить наилучший стиль или способ описания для всех конечных автоматов. Обычно стиль и способ описания конечного автомата выбирается на основании особенностей автомата, условий использования, а также требований, предъявляемых к конечному автомату.

В данной статье рассматриваются некоторые полезные стили и способы описания конечных автоматов, которые разработчики не всегда применяют на практике. Для описания проектов мы будем использовать язык Verilog [3], но все остается справедливым и для языка SystemVerilog [4]. Рассматриваемые примеры конечных автоматов были реализованы в системе Quartus Prime от Intel версии 18.1 в FPGA семейства Cyclone IV E, а моделирование выполнялось с помощью упрощенной версии временного моделирования University Program в редакторе временных диаграмм Simulation Waveform Editor. Все установки Синтезатора и Упаковщика (Fitter) были приняты по умолчанию. Для рассматриваемых примеров использовался следующий SDC-файл временных ограничений:

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

create_clock -name virt_clk_in -period 10.000

create_clock -name virt_clk_out -period 10.000

derive_pll_clocks -create_base_clocks

derive_clock_uncertainty

set_input_delay -clock { virt_clk_in } -min 0
[get_ports {reset x[*] }]

set_input_delay -clock { virt_clk_in } -max 1
[get_ports {reset x[*] }]

set_output_delay -clock { virt_clk_out } -min 0
[get_ports { y[*] }]

set_output_delay -clock { virt_clk_out } -max 1
```

Статья организована следующим образом. В разделе 2 приводятся традиционные описания конечных автоматов Мили и Мура на языке Verilog с 3 процессами. Вопросы описания конечных автоматов с регистрами на выходах рассматриваются в разделе 3. Исследование оператора

case для описания условий переходов из каждого состояния представлено в разделе 4. Стили описания надежных (безопасных – safe) конечных автоматов даются в разделе 5. В разделе 6 рассматриваются способы проверки значений входных сигналов. Раздельное и векторное определение значений выходных сигналов исследуется в разделе 7. Определение значений по умолчанию для состояния перехода и выходных сигналов, а также одновременно для состояния перехода и выходных сигналов приводится в разделах 8, 9 и 10. Анализ рассмотренных способов описания конечных автоматов выполняется в разделе 11. Итоги подводятся в разделе 12.

2. Традиционное описание конечных автоматов на языке Verilog

Традиционным считается представление конечного автомата с тремя процессами [5], когда с помощью трех операторов **always** отдельно описывается память конечного автомата, функции переходов и функции выходов. Мы будем использовать буквы M, T и O для обозначения процессов, которые описывают память (memory – M), переходы (transitions – T) и выходы (outputs – O) конечного автомата. В листинге 1 приводится традиционное описание конечного автомата Мили с тремя процессами.

Результаты синтеза и моделирования проекта Mealy_3proc_M_T_O показаны на рис. 1 и 2.

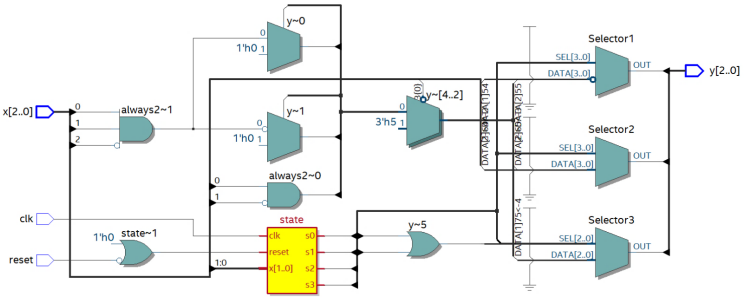


Рис. 1. Результаты синтеза проекта Mealy_3proc_M_T_O на уровне регистровых передач

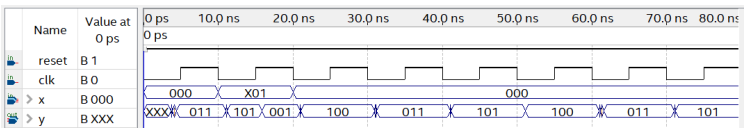


Рис. 2. Результаты временного моделирования проекта Mealy_3proc_M_T_O

В листинге 2 приводится традиционное описание конечного автомата Мура, который реализует тот же алгоритм логического управления.

```
module Mealy_3proc_M_T_O (
 input clk, reset,  // сигналы синхронизации и сброса
 input [2:0] x,   // входные переменные
 output reg [2:0] y); // выходные переменные

 reg [1:0]  state,   // настоящее состояние автомата
            next;   // следующее состояние автомата

 localparam [1:0]  // объявление состояний
  s0=2'b00,   // (начальное состояние)
  s1=2'b01,   // с указанием их кодов
  s2=2'b10,
  s3=2'b11;

// процесс M, описание памяти автомата (регистра состояний)
 always @(posedge clk, negedge reset)
  if (~reset)  state <= s0;
  else  state <= next;

// процесс T, описание переходов
 always @(*)
  case (state)
   s0:      next = s1;
   s1: if (~x[0])      next = s2;
    else if (~x[1] && x[0])  next = s3;
    else if (~x[2] && x[1] && x[0]) next = s0;
    else       next = s0;
   s2: if (~x[2])      next = s0;
    else      next = s0;
   s3:      next = s0;
  endcase

// процесс O, описание выходов
 always @(*)
  case (state)
   s0:      y = 3'b011;
   s1: if (~x[0])      y = 3'b101;
    else if (~x[1] && x[0])  y = 3'b001;
    else if (~x[2] && x[1] && x[0]) y = 3'b100;
    else       y = 3'b010;
   s2: if (~x[2])      y = 3'b100;
    else      y = 3'b010;
   s3:      y = 3'b100;
  endcase
endmodule
```

Листинг 1. Проект Mealy_3proc_M_T_O. Традиционное описание автомата Мили с 3 процессами.

Результаты синтеза и моделирования проекта Moore_ 3_proc_M_T_O_case показаны на рис. 3 и 4.

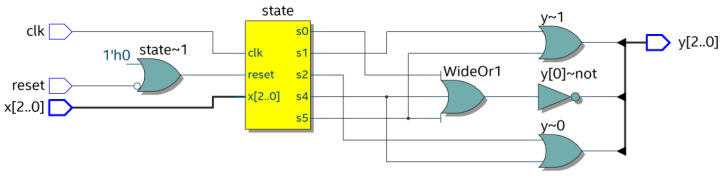


Рис. 3. Результаты синтеза проекта Moore_ 3_proc_M_T_O_case

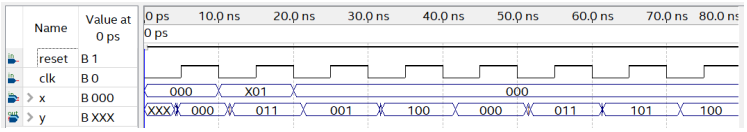


Рис. 4. Результаты временного моделирования проекта Moore_ 3_proc_M_T_O_case

Существенным отличием приведенного кода от аналогичного описания автомата Мили является определение выходов. В нашем коде для этого используется оператор **case**, который проверяет в каком состоянии находится конечный автомат и формирует соответствующий выходной вектор. С этой целью может также использоваться оператор **if** [3].

Стили описания конечных автоматов Мили и Мура с 2 процессами и с 1 процессом рассматриваются в [5]. Однако эти стили описания не позволяют существенно изменить свойства и параметры конечных автоматов.

3. Описание конечных автоматов с регистрами на выходах

Иногда бывает необходимо специально установить регистры на выходах клнечного автомата, например, для лучшей привязки времени формирования выходных сигналов к тактам синхросигнала. Для этого достаточно в процессе O, который описывает функции выходов, вместо **always @(*)** записать **always @(posedge** clk**)**.

Кроме того, регистр в структуре конечного автомата можно описать явно. Это достаточно просто сделать путем объявления дополнительных промежуточных переменных для выходов конечного автомата и описать выходной регистр с помощью еще одного оператора **always** (листинг 3).

```
module Moore_3_proc_M_T_O_case (
 input clk,reset,
 input [2:0] x,
 output reg [2:0] y);

 reg [2:0] state, next;     // переменные состояний

 localparam [2:0]     // описание состояний
  s0=3'b000,
  s1=3'b001,
  s2=3'b010,
  s3=3'b011,
  s4=3'b100,
  s5=3'b101;

 // процесс M, описание памяти автомата
 always @(posedge clk, negedge reset)
  if (~reset)  state <= s0;
  else  state <= next;

 // // процесс T, описание переходов
 always @(*)
  case (state)
   s0:       next = s1;
   s1: if (~x[0])      next = s2;
    else if   (~x[1] && x[0])   next = s3;
    else if   (~x[2] && x[1] && x[0]) next = s4;
    else       next = s5;
   s2: if (~x[2])      next = s4;
    else      next = s5;
   s3:       next = s4;
   s4:       next = s0;
   s5:       next = s0;
  endcase

 // процесс O, описание выходов
 always @(*)
  case (state)
   s0: y = 3'b000;
   s1: y = 3'b011;
   s2: y = 3'b101;
   s3: y = 3'b001;
   s4: y = 3'b100;
   s5: y = 3'b010;
  endcase
endmodule
```

Листинг 2. Проект Moore_ 3_proc_M_T_O_case. Традиционное описание автомата Мура с 3 процессами и оператором **case** для описания выходов

```
module Mealy_3proc_M_T_O_with_output_register(
 input clk,reset,
 input [2:0] x,
 output reg [2:0] y_out); // выходы автомата

 reg [2:0] y;   // промежуточная переменная

 … // объявления состояний и переменных состояний как
в листинге 1
 … // описание регистра состояний как в листинге 1
 … // описание функций переходов как в листинге 1
 … // описание функций выходов как в листинге 1

 always @(posedge clk, negedge reset) // выходной
регистр
  if (~reset)  y_out <= 0;
  else  y_out <= y;
```

Листинг 3. Проект Mealy_3proc_M_T_O_with_output_register. Проект автомата Мили с выходным регистром.

Результаты синтеза и моделирования проекта Mealy_3proc_M_T_O_with_output_register показаны на рис. 5 и 6.

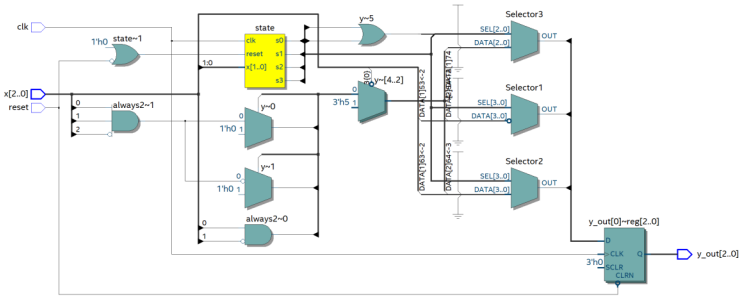


Рис. 5. Результаты синтеза проекта Mealy_3proc_M_T_O_with_output_register

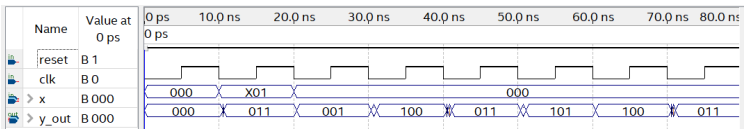


Рис. 6. Результаты временного моделирования проекта Mealy_3proc_M_T_O_with_output_register

Из рис. 5 видно, что на выходах схемы конечного автомата установлен регистр.

4. Использование оператора case в описании конечных автоматов

В рассматриваемых до сих пор кодах конечных автоматов условия переходов из каждого состояния проверялись с помощью операторов **if**. Благодаря взаимозаменяемости операторов **if** и **case** условия переходов из каждого состояния можно также описать с помощью оператора **case**. В листинге 4 приводится описание проекта Mealy_3proc_M_T_O_case с использованием оператора **case** для проверки условий переходов из каждого состояния.

```
module Mealy_3proc_M_T_O_case (…);

... // объявления, повторение проекта
Mealy_3proc_M_T_O

 always @(posedge clk, negedge reset)
  if (~reset)  state <= s0;
  else  state <= next;

 always @(*)
  case (state)
   s0:    next = s1;
   s1: casex (x)
     3'b??0:   next = s2;
     3'b?01:   next = s3;
     3'b011:  next = s0;
     3'b111:  next = s0;
    endcase
   s2: casex (x)
     3'b0??:   next = s0;
     3'b1??:   next = s0;
    endcase
   s3:    next = s0;
  endcase

 always @(*)
  case (state)
   s0:    y = 3'b011;
   s1: casex (x)
     3'b??0:   y = 3'b101;
     3'b?01:  y = 3'b001;
     3'b011:  y = 3'b100;
     3'b111:  y = 3'b010;
    endcase

   s2: casex (x)
     3'b0??:   y = 3'b100;
     3'b1??:   y = 3'b010;
    endcase
   s3:    y = 3'b100;
  endcase
endmodule
```

Листинг 4. Проект Mealy_3proc_M_T_O_case, использование оператора **case** для проверки условий переходов из каждого состояния

Отметим, что в листинге 4 для проверки значений входных векторов используются операторы **casex**, поскольку константные элементы оператора **case** [3] могут содержать знак вопроса («?»), который соответствует неизвестному или произвольному значению бита входного вектора.

Результаты синтеза проекта Mealy_3proc_M_T_O_case показаны на рис. 7, которые во многом подобны результатам синтеза проекта Mealy_3proc_M_T_O (рис. 1). Результаты моделирования проекта Mealy_3proc_M_T_O_case полностью совпадают с результатами моделирования проекта Mealy_3proc_M_T_O (рис. 2).

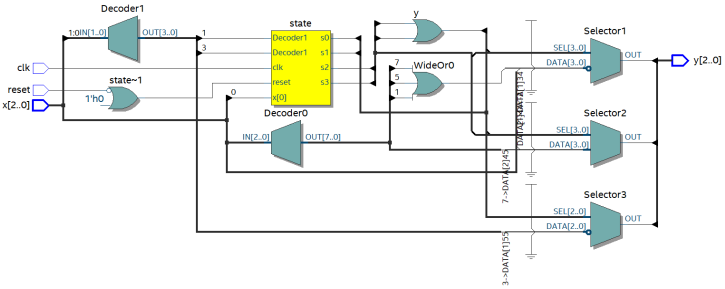


Рис. 7. Результаты синтеза проекта Mealy_3proc_M_T_O_case

Подобным образом оператор **case** может использоваться во всех описаниях конечных автоматов, как Мили, так и Мура.

5. Описание надежных конечных автоматов

Корректным стилем описания конечного автомата является такое описание, когда

- для каждого состояния описываются все возможные переходы из данного состояния;
- логическая сумма всех условий переходов из одного состояния равна логической единице;
- в каждое состояние ведет по крайней мере один переход из других состояний;
- из каждого состояния по крайней мере один переход ведет в другие состояния.

При таком описании в автомате не должны возникать недопустимые переходы или условия переходов, которые не инициируют никаких переходов.

Сложнее дело обстоит с кодами состояний. Если в результате сбоя в регистре памяти конечного автомата сформируется код, который не соответствует ни одному внутреннему состоянию конечного автомата, автомат постоянно будет находиться в недопустимом (illegal) состоянии.

Следующий стиль позволяет повысить надежность функционирования конечного автомата. Для этого модифицируем стиль проекта Mealy_3proc_M_T_O следующим образом. В операторе **case** в случае не совпадения значения исходного состояния перехода (переменная *state*) ни с одним из определенных состояний, опишем возврат в начальное состояние s0 с помощью конструкции **default**, а для выходов – формирование нулевого вектора. Такой стиль описания надежного автомата назовем *reliability1* (листинг 5).

```
module Mealy_3proc_M_T_O_reliability1 (…);  // порты автомата как в листинге 1
 … // объявление переменных и состояний как в листинге 1
 … // объявление памяти автомата как в листинге 1

 always @(*)        // описание переходов
  case (state)
   s0:       next = s1;
   s1: if (~x[0])      next = s2;
    else if (~x[1] && x[0])  next = s3;
    else if (~x[2] && x[1] && x[0]) next = s0;
    else       next = s0;
   s2: if (~x[2])      next = s0;
    else      next = s0;
   s3:       next = s0;
   default:       next = s0;// возврат в s0
  endcase

 always @(*)        // описание выходов
  case (state)
   s0:       y = 3'b011;
   s1: if (~x[0])      y = 3'b101;
    else if (~x[1] && x[0])  y = 3'b001;
    else if (~x[2] && x[1] && x[0]) y = 3'b100;
    else       y = 3'b010;
   s2: if (~x[2])      y = 3'b100;
    else      y = 3'b010;
   s3:       y = 3'b100;
   default:       y = 3'b000;// нулевой вектор
  endcase
endmodule
```

Листинг 5. Проект Mealy_3proc_M_T_O_reliability1, описание «надежного» конечного автомата

Результаты синтеза и моделирования проекта Mealy_3proc_M_T_O_reliability1 полностью совпадают с аналогичными результатами проекта Mealy_3proc_M_T_O (рис. 1 и 2).

Можно еще более повысить надежность функционирования конечного автомата путем специального описания. Для этого модифицируем стиль *reliability1* следующим образом. В случае не совпадения значения исходного состояния перехода (переменная *state*) ни с одним из определенных состояний, выполняется возврат в начальное состояние s0 (как в стиле reliability1); в случае не выполнения ни одного из условий перехода из исходного состояния перехода, автомат остается в исходном состоянии перехода (т.е. реализуется состояние ожидания). Такой стиль описания конечных автоматов назовем *reliability2* (листинг 6).

Результаты синтеза и моделирования проекта Mealy_3proc_M_T_O_reliability2 полностью совпадают с аналогичными результатами проекта Mealy_3proc_M_T_O_case (рис. 7 и 2).

В системе Quartus имеется специальный режим синтеза, который позволяет строить *безопасные конечные автоматы* (Safe State Machines). По умолчанию режим синтеза безопасных конечных автоматов выключен. Чтобы включить опцию синтеза безопасного конечного автомата, достаточно выбрать Assignment > Settings > Compiler Settings > Advanced Settings (Synthesis) и установить параметр Safe State Machines в значение On. При включении данного режима программное обеспечение вставляет в проект дополнительную логику для обнаружения *недопустимых состояний* (illegal states) конечного автомата и реализации безусловного перехода в начальное состояние из каждого недопустимого состояния.

Следует отметить, что включение опции Safe State Machines может привести к значительному увеличению стоимости реализации конечного автомата. Обычно опцию Safe State Machines включают, когда управляющие сигналы конечного автомата прибывают из другого домена (области) синхронизации, а проект имеет асинхронные входные данные. Однако Intel рекомендует в таких случаях вместо опции Safe State Machines использовать цепочку регистров синхронизации для привязки сигналов к синхросигналу конечного автомата.

В наших описаниях конечных автоматов безопасному конечному автомату соответствует стиль reliability1.

```
module Mealy_3proc_M_T_O_reliability2 (…);  // порты автомата как в листинге 1
 … // объявление переменных и состояний как в листинге 1
 … // объявление памяти автомата как в листинге 1

 always @(*)      // описание переходов
  case (state)
   s0:    next = s1;
   s1: casex (x)
     3'b??0:   next = s2;
     3'b?01:   next = s3;
     3'b011:  next = s0;
     3'b111:  next = s0;
     default:  next = s1; // цикл ожидания
    endcase
   s2: casex (x)
     3'b0??:   next = s0;
     3'b1??:   next = s0;
     default:  next = s2; // цикл ожидания
    endcase
   s3:    next = s0;
   default:    next = s0; // возврат в s0
  endcase

 always @(*)      // описание выходов
  case (state)
   s0:    y = 3'b011;
   s1: casex (x)
     3'b??0:   y = 3'b101;
     3'b?01:   y = 3'b001;
     3'b011:  y = 3'b100;
     3'b111:  y = 3'b010;
     default:  y = 3'b000; // нулевой вектор
    endcase

   s2: casex (x)
     3'b0??:   y = 3'b100;
     3'b1??:   y = 3'b010;
     default:  y = 3'b000; // нулевой вектор
    endcase
   s3:    y = 3'b100;
   default:    y = 3'b000; // нулевой вектор
  endcase
endmodule
```

Листинг 6. Проект Mealy_3proc_M_T_O_reliability2, еще одно описание «надежного» конечного автомата

6. Раздельная и векторная проверка значений входных сигналов

Значения входных сигналов конечных автоматов проверяются в описании функций переходов, а также функций выходов для автоматов типа Мили. Значения входных сигналов можно проверять раздельно или все вместе в виде входного вектора.

Раздельная проверка значений входных сигналов применяется в традиционных описаниях автомата Мили (листинг 1) и автомата Мура (листинг 2), а также в других описаниях, представленных выше.

Значения входных сигналов можно также проверять все вместе, т.е. одновременно, путем проверки значения входного вектора. Проверка значений входных сигналов в виде входного вектора для автомата Мили приведена в листинге 4, а для автомата Мура – в листинге 7.

```
module Moore_3proc_M_T_O_input_vector(…); // объявление портов как в листинге 2
 … // объявление переменных и состояний как в листинге 2
 … // описание памяти автомата как в листинге 2

 always @(*)      // описание переходов
  case (state)
   s0:    next = s1;
   s1: casex(x)
     3'b??0:   next = s2;
     3'b?01:   next = s3;
     3'b011: next = s4;
     3'b111: next = s5;
    endcase
   s2: casex(x)
     3'b0??:   next = s4;
     3'b1??:   next = s5;
    endcase
   s3:    next = s4;
   s4:    next = s0;
   s5:    next = s0;
  endcase

 … // описание выходов как в листинге 2
endmodule
```

Листинг 7. Проект Moore_3proc_M_T_O_input_vector, проверка значений входных сигналов в виде входного вектора.

Результаты синтеза проекта Moore_3proc_M_T_O_input_vector (рис. 8) во многом подобны результатам синтеза проекта Moore_3proc_M_T_O_case на рис. 3, а результаты моделирования полностью совпадают с результатами временного моделирования проекта Moore_3proc_M_T_O_case на рис. 4.

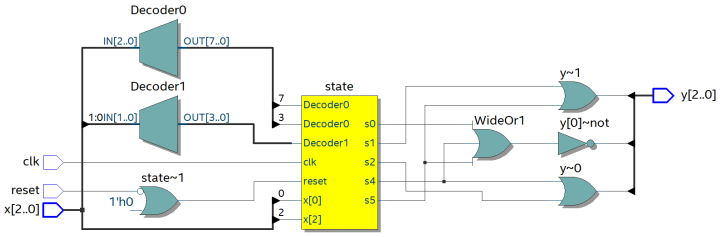


Рис. 8. Результаты синтеза на RTL-уровне проекта Moore_3proc_M_T_O_input_vector

Вывод. Раздельная или векторная проверка значений входных сигналов конечных автоматов, как для автомата Мили, так и для автомата Мура, оказывает незначительное влияние на результаты синтеза и совершенно не влияет на результаты моделирования.

7. Раздельное и векторное определение значений выходных сигналов

Формирование значений выходных сигналов конечного автомата, по аналогии с проверкой значений входных сигналов, можно выполнять либо путем присвоения значения всему выходному вектору, либо присваивая значения каждому выходному сигналу по отдельности. Во всех рассматриваемых ранее листингах значения выходным сигналам назначались путем присвоения значения всему выходному вектору. Иногда бывает удобно значения каждому выходному сигналу присваивать по отдельности.

Рассмотрим векторное и раздельное присваивание значений выходным сигналам конечных автоматов на примере автомата Мура из нашего примера. Векторное присвоение значений выходным сигналам для автомата Мура приведено в листинге 2, а результаты синтеза и моделирования показаны на рис. 3 и 4. Раздельное присвоение значений выходным сигналам для автомата Мура приведено в листинге 8.

Результаты синтеза и моделирования проекта Moore_3proc_M_T_O_separate_outputs полностью совпадают с аналогичными результатами проекта Moore_3proc_M_T_O_case (рис. 3 и 4).

Польза от раздельного присвоения значений выходных сигналов становится заметной, если вначале блока **always** всем выходным сигналами присвоить некоторое

```
module Moore_3proc_M_T_O_separate_outputs(
…); // объявление портов как в листинге 2
…// объявление переменных и состояний как в листинге 2
 …// описание памяти автомата как в листинге 2
 …// описание переходов как в листинге 2

 always @(*)  // раздельное определение значений
выходных сигналов
  case (state)
   s0: begin y[2]=1'b0; y[1]=1'b0; y[0]=1'b0; end
   s1: begin y[2]=1'b0; y[1]=1'b1; y[0]=1'b1; end
   s2: begin y[2]=1'b1; y[1]=1'b0; y[0]=1'b1; end
   s3: begin y[2]=1'b0; y[1]=1'b0; y[0]=1'b1; end
   s4: begin y[2]=1'b1; y[1]=1'b0; y[0]=1'b0; end
   s5: begin y[2]=1'b0; y[1]=1'b1; y[0]=1'b0; end
  endcase
endmodule
```

Листинг 8. Проект Moore_3proc_M_T_O_separate_outputs, раздельное определение значений выходных сигналов.

значение по умолчанию, а затем для каждого состояния (в случае автомата Мура) или на каждом переходе (в случае автомата Мили) присваивать значения только для тех сигналов, которые должны иметь значения отличные от значения по умолчанию. При таком описании повышается читабельность исходного кода, поскольку из кода проекта хорошо видно, какие выходные сигналы изменяют свое значение в данном состоянии или на данном переходе.

В общем случае значения по умолчанию для выходных сигналов могут быть любыми. Для конечного автомата Мура из нашего примера по умолчанию всем выходным сигналам присвоим нулевые значения (листинг 9).

Результаты синтеза проекта Moore_3proc_M_T_O_separate_outputs_with_default приведены на рис. 9, а результаты моделирования остались прежними (рис. 4).

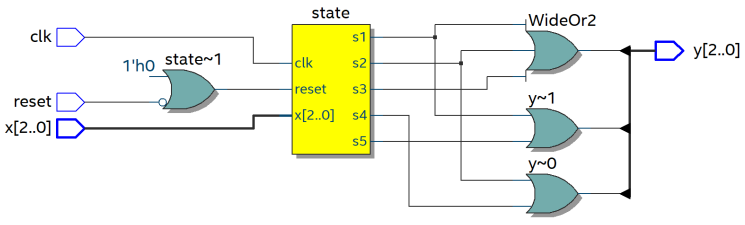


Рис. 9. Результаты синтеза на RTL-уровне проекта Moore_3proc_M_T_O_separate_outputs_with_default

Из рис. 9 видно, что немного изменилась логика формирования выходных сигналов, по сравнению с рис. 3. Кроме того, немного увеличилась максимальная частота функционирования автомата с 611 до 640 MHz.

```
module Moore_3proc_M_T_O_separate_outputs_with_default(
 …);// объявление портов как в листинге 2
…// объявление переменных и состояний как в листинге 2
 …// описание памяти автомата как в листинге 2
 …// описание переходов как в листинге 2

 always @(*) begin    // описание выходов
  y = 3'b000;    // значение по умолчанию
  case (state)
   s1: begin y[1]=1'b1; y[0]=1'b1; end
   s2: begin y[2]=1'b1; y[0]=1'b1; end
   s3: y[0]=1'b1;
   s4: y[2]=1'b1;
   s5: y[1]=1'b1;
  endcase
 end
endmodule
```

Листинг 9. Проект Moore_3proc_M_T_O_separate_outputs_with_default, раздельное определение значений выходных сигналов со значениями по умолчанию.

Вывод. Раздельное присваивание значений выходных сигналов при использовании значений по умолчанию в некоторых случаях позволяет упростить логику формирования выходных сигналов и, как следствие, увеличить быстродействие конечного автомата. Кроме того, при раздельном присваивании значений выходным сигналам повышается читабельность исходного кода проекта.

8. Определение значения по умолчанию для состояния перехода

Состоянием перехода (next state) конечного автомата называется состояние, в которое переходит автомат в следующем такте синхронизации. В приводимых описаниях конечных автоматов состояние перехода определяется переменной *next*. Идея определения значения по умолчанию для состояния перехода заключается в том, что вначале блока **always**, который описывает переходы конечного автомата, переменной *next* присваивается значение по умолчанию, а затем определяются только такие значения переменной *next*, которые отличаются от значения по умолчанию.

В качестве значения по умолчанию следует выбирать состояние конечного автомата, в которое ведет максимальное число переходов с наиболее сложными условиями переходов. Сложность условий переходов для синтезируемого конечного автомата можно определить по списку переходов, который формирует Компилятор средства проектирования.

Рассмотрим традиционное описание конечного автомата Мили в листинге 1. Здесь наибольшее число переходов выполняется в состояние s0 и условия переходов в состояние s0 достаточно сложные. Поэтому в качестве значения по умолчанию для переменной *next* выбираем состояние s0 (листинг 10).

```
module Mealy_3proc_M_T_O_next_default(
 …);       // объявление портов

…// объявление переменных состояния как в листинге 1
 …// объявление состояний как в листинге 1
 …// описание памяти как в листинге 1

 always @(*) begin     // описание переходов
  next = s0;  // значение по умолчанию для состояния перехода
  case (state)   // здесь удалены все переходы в состояние s0
   s0:       next = s1;
   s1: if (~x[0])      next = s2;
    else if (~x[1] && x[0])  next = s3;
  endcase
 end

…// описание выходов как в листинге 1
endmodule
```

Листинг 10. Проект Mealy_3proc_M_T_O_next_default, определение значения по умолчанию для состояния перехода.

В результате синтеза проекта Mealy_3proc_M_T_O_next_default стоимость реализации не изменилась (L= 9), однако значительно возросло быстродействие конечного автомата (Fmax = 1136 MHz по сравнению с 305 MHz).

Результаты синтеза проекта Mealy_3proc_M_T_O_next_default полностью совпадают с результатами синтеза проекта Mealy_3proc_M_T_O (рис. 1).

Вопрос: почему значительно увеличилось быстродействие конечного автомата? Ответ на этот вопрос дает сравнение технологических карт проекта Mealy_3proc_M_T_O (рис. 10) и проекта Mealy_3proc_M_T_O_next_default (рис. 11).

Еще большее различие для этих проектов наблюдается в технологических картах после упаковки (Post-Fitting Technology Map). Другими словами, значительное увеличение быстродействия проекта Mealy_3proc_M_T_O_next_default, по сравнению с проектом Mealy_3proc_M_T_O, произошло в результате работы Упаковщика (Fitter), алгоритм работы которого нам неизвестен.

Результаты моделирования проекта Mealy_3proc_M_T_O_next_default (рис. 12) во многом совпадают с результатами моделирования проекта Mealy_3proc_M_T_O (рис. 2), однако увеличились временные интервалы с неопределенными значениями. В некоторых случаях это может вызвать серьёзные

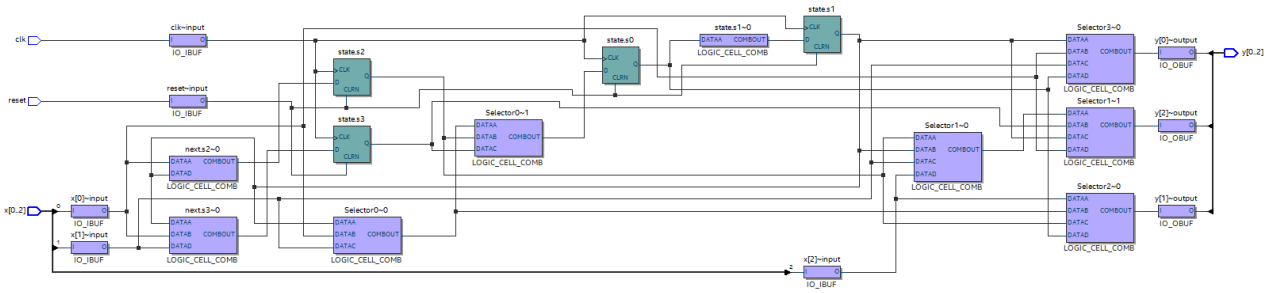


Рис. 10. Технологическая карта проекта Mealy_3proc_M_T_O

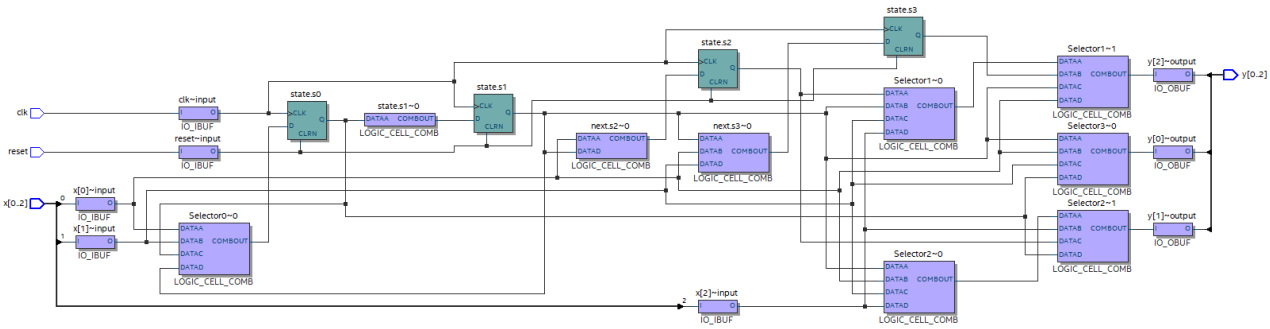


Рис. 11. Технологическая карта проекта Mealy_3proc_M_T_O_next_default

проблемы и необходимость установки регистра на выходе конечного автомата.

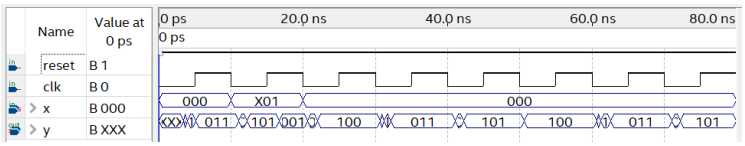


Рис. 12. Результаты временного моделирования проекта Mealy_3proc_M_T_0_next_default

В листинге 11 значение по умолчанию переменной *next* присваивается не вначале блока **always**, а с помощью конструкции **default** оператора **case**.

```
module Mealy_3proc_M_T_0_next_default_2(
 …);       // объявления портов

…// объявление переменных состояния как в листинге 1
 …// объявление состояний как в листинге 1
 …// описание памяти как в листинге 1

 always @(*)       // описание переходов
  case (state)
   s0:       next = s1;
   s1: if (~x[0])      next = s2;
    else if (~x[1] && x[0])  next = s3;
   default: next = s0;  // значение по умолчанию
  endcase

…// описание выходов как в листинге 1
endmodule
```

Листинг 11. Проект Mealy_3proc_M_T_0_next_with_default_2, определение значения по умолчанию для состояния перехода с помощью конструкции **default** оператора **case**.

Напомним, в блоке **always** все операторы выполняются последовательно. Поэтому в листинге 10 значение по умолчанию переменной *next* присваивается вначале блока **always** и при выполнении последующих операторов может быть изменено. Если в цепочке операторов **if-else-if** указаны не все возможные значения входов, это не вызывает проблем, поскольку значение переменной *next* уже установлено.

В листинге 11 значения переменной *next* определены не для всех возможных значений входов. Поэтому Компилятор установит защелки для запоминания последнего значения переменной *next*. Кроме того, в листинге 11 переходы определены не для всех состояний, в связи с этим Компилятор не может построить список переходов конечного автомата.

В результате синтеза проекта Mealy_3proc_M_T_0_next_default_2 стоимость реализации возросла (L = 13) из-за введения защелок, временной анализатор не смог определить максимальную частоту функционирования (Fmax) из-за ошибок, связанных с комбинационными циклами. Однако результаты моделирования не изменились и совпадают с рис. 2.

Выводы. Определение значения по умолчанию для состояния перехода в некоторых случаях позволяет уменьшить код проекта и увеличить быстродействие конечного автомата, однако для автоматов Мили увеличиваются временные интервалы с неопределенными значениями на выходе.

Значение по умолчанию для состояния перехода следует присваивать вначале блока **always**; не следует с этой целью использовать конструкцию **default** оператора **case**.

9. Определение значений по умолчанию выходных сигналов

Применим концепцию определения значений по умолчанию к выходным сигналам на примере конечного автомата Мили, когда значения присваиваются всему выходному вектору. Отметим, что данный подход в случае раздельного присваивания значений выходным сигналам в случае автомата Мура рассмотрен в листинге 9.

В качестве выходного вектора по умолчанию выберем вектор, который наиболее часто формируется на переходах между состояниями и условия переходов являются наиболее сложными. Для конечного автомата Мили из нашего примера таким вектором является вектор «100». Значение выходного вектора по умолчанию присваивается в начале блока **always**, а затем определяются только такие выходные векторы, которые отличаются от значения по умолчанию (листинг 12).

Результаты синтеза проекта Mealy_3proc_M_T_0_outputs_default (рис. 13) во многом совпадают с результатами синтеза проекта Mealy_3proc_M_T_0 (рис. 1), однако стоимость реализации (L = 9) и быстродействие (F = 305) не изменились. Результаты моделирования проекта Mealy_3proc_M_T_0_outputs_default полностью совпадают с результатами моделирования проекта Mealy_3proc_M_T_0 (рис. 2).

```
module Mealy_3proc_M_T_O_outputs_default(
 …);       // объявление портов

…// объявление переменных состояния как в листинге 1
 …// объявление состояний как в листинге 1
 …// описание памяти как в листинге 1
 …// описание переходов как в листинге 1

 always @(*) begin     // описание выходов
  y = 3'b100;  // определение значения по умолчанию
  case (state)
   s0:       y = 3'b011;
   s1: if (~x[0])      y = 3'b101;
    else if (~x[1] && x[0])   y = 3'b001;
    else if ( x[2] && x[1] && x[0]) y = 3'b010;
   s2: if (x[2])      y = 3'b010;
  endcase
 end
endmodule
```

Листинг 12. Проект Mealy_3proc_M_T_O_outputs_default определение значения по умолчанию для выходных векторов

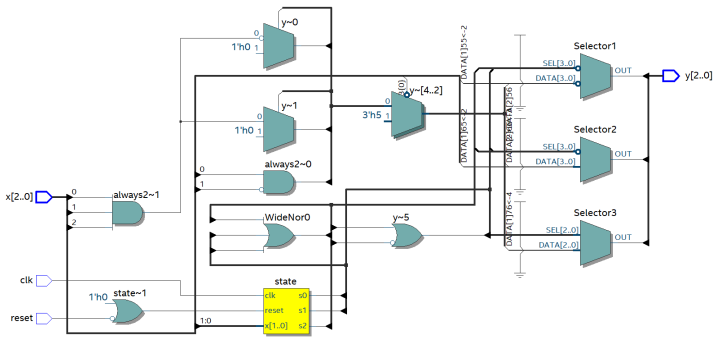


Рис. 13. Результаты синтеза проекта Mealy_3proc_M_T_O_outputs_default

Можно предположить, что при частом повторении одного и того же выходного вектора на различных переходах между состояниями рассмотренный подход описания конечных автоматов приведет к сокращению исходного кода, а также к уменьшению стоимости реализации и увеличению быстродействия. В нашем примере это предположение не подтвердилось из-за малых размеров конечного автомата.

Вывод. Определение значений по умолчанию выходных сигналов следует использовать при частом повторении одного и того же выходного вектора на различных переходах (для автомата Мили) или в различных состояниях (для автомата Мура). Это позволяет сократить исходный код проекта, а также в некоторых случаях может привести к уменьшению стоимости реализации и увеличению быстродействия.

10. Определение значений по умолчанию для состояния перехода и выходных сигналов

В листинге 13 представлено описание конечного автомата Мили из нашего примера, когда значение по умолчанию определено как для состояния перехода, так и для выходных сигналов.

Результаты синтеза проекта

```
module Mealy_3proc_M_T_O_next_outputs_default(
 …);       // описание портов

…// объявление переменных состояния как в листинге 1
 …// объявление состояний как в листинге 1
 …// описание памяти как в листинге 1

 always @(*) begin     // описание переходов
  next = s0;  // значение по умолчанию
  case (state)
   s0:       next = s1;
   s1: if (~x[0])      next = s2;
    else if (~x[1] && x[0])   next = s3;
  endcase
 end

 always @(*) begin     // описание выходов
  y = 3'b100;  // значение по умолчанию
  case (state)
   s0: y = 3'b011;
   s1: if (~x[0])      y = 3'b101;
    else if (~x[1] && x[0])   y = 3'b001;
    else if ( x[2] && x[1] && x[0]) y = 3'b010;
   s2: if (x[2])      y = 3'b010;
  endcase
 end
endmodule
```

Листинг 13. Проект Mealy_3proc_M_T_O_next_outputs_default, определение значений по умолчанию для состояния перехода и выходных векторов.

Mealy_3proc_M_T_O_next_outputs_default полностью совпадают с результатами синтеза проекта Mealy_3proc_M_T_O_outputs_default (рис. 13), а результаты моделирования – с результатами моделирования проекта Mealy_3proc_M_T_O_next_default (рис. 12).

Вывод. В одном описании конечного автомата можно одновременно использовать значения по умолчанию как для состояния перехода, так и для выходных сигналов.

Отметим, что все рассмотренные способы описания конечных автоматов могут также использоваться в случае описания конечных автоматов на языке SystemVerilog.

11. Анализ рассмотренных способов описания конечных автоматов на языках Verilog и SystemVerilog

В табл. 1 суммируются результаты рассмотренных способов описания конечных автоматов из нашего примера, где L – число используемых логических элементов (стоимость реализации или площадь); F – максимальная частота в мегагерцах (быстродействие); P – число строк исходного кода.

Из табл. 1 видно, что рассмотренные способы описания конечных автоматов не влияют на стоимость реализации проектов. Исключением является проект Mealy_3proc_M_T_0_next_default_2, который не рекомендован к использованию.

Быстродействие автомата Мура увеличивается в случае раздельного присваивания значений выходным сигналам с заданием значений по умолчанию (проект Moore_3proc_M_T_0_separate_outputs_with_default). Быстродействие автомата Мили значительно увеличивается в случае определения значения по умолчанию для состояния перехода (проекты Mealy_3proc_M_T_0_next_default и Mealy_3proc_M_T_0_next _outputs_default).

Число P строк исходного кода для различных проектов изменяется от 38 до 50 и может рассматриваться как важный параметр только для очень больших проектов. Наибольшее значение параметра P наблюдается в проекте Mealy_3proc_M_T_0_case при проверке значений входного вектора с помощью оператора **case**. Наименьшее значение параметра P наблюдается в проектах, где задаются значения по умолчанию для состояния переходов и выходных векторов.

Общий вывод. Из всех рассмотренных способов описания конечных автоматов для практического использования можно рекомендовать раздельное и, возможно, векторное присвоение значений выходным сигналам со значениями по умолчанию (проект Moore_3proc_M_T_0_separate_outputs_with_default), а также определение значения по умолчанию для состояния перехода (проекты Mealy_3proc_M_T_0_next_default и Mealy_3proc_M_T_0_next_outputs_default), так как эти способы в некоторых случаях позволяют увеличить быстродействие конечного автомата, улучшить читабельность исходного кода и способствуют уменьшению стоимости реализации проекта.

Таблица 1. Результаты различных способов описания конечного автомата Мили на языке SystemVerilog

Листинг	Проект	L	F	P	Примечания
1	Mealy_3proc_M_T_0	9	305	42	Раздельная проверка значений входных сигналов
2	Moore_3proc_M_T_0_case	10	611	44	Раздельная проверка значений входных сигналов, векторное присвоение значений выходным сигналам
3	Mealy_3proc_M_T_0_with_output_register	9	388	47	Явное описание выходного регистра
4	Mealy_3proc_M_T_0_case	9	305	50	Использование оператора case для проверки значения входного вектора
5	Mealy_3proc_M_T_0_reliability1	9	305	44	Описание надежного автомата в стиле reliability1
6	Mealy_3proc_M_T_0_reliability2	9	305	56	Описание надежного автомата в стиле reliability2
7	Moore_3proc_M_T_0_input_vectors	10	611	48	Проверка значения входного вектора
8	Moore_3proc_M_T_0_separate_outputs	10	611	44	Раздельное присвоение значений выходным сигналам
9	Moore_3proc_M_T_0_separate_outputs _with_default	10	640	44	Раздельное присвоение значений выходным сигналам со значениями по умолчанию
10	Mealy_3proc_M_T_0_next_default	9	1136	39	Определение значения по умолчанию для состояния перехода
11	Mealy_3proc_M_T_0_next_default_2	13	-	38	Определение значения по умолчанию для состояния перехода с помощью конструкции default оператора case
12	Mealy_3proc_M_T_0_outputs_default	9	305	40	Определение значения по умолчанию для выходных векторов
13	Mealy_3proc_M_T_0_next_outputs_default	9	1136	38	Определение значения по умолчанию для состояния перехода и выходных векторов

Выбор между раздельной и векторной проверкой входных сигналов должен делать разработчик, исходя из целей описания конечного автомата.

Следует также отметить, что в описаниях конечных автоматов важны как стили описания, так и способы описания на языках Verolog и SystemVerilog. Например, для конечного автомата Мили из нашего примера использование способа описания, который определяет значение по умолчанию для состояния перехода, позволило увеличить быстродействие с 305 MHz до 1136 MHz, т.е. в 3,72 раза. По мнению автора, такое увеличение быстродействия конечного автомата сложно сделать как с помощью методов кодирования состояний [6, 7], так и путем использования специальных методов синтеза [8].

12. Выводы

Каждый стиль и способ описания имеет свои положительные и отрицательные стороны, поэтому нельзя выделить наилучший стиль или способ описания для всех конечных автоматов. Обычно стиль и способ описания конечного автомата выбирается на основании особенностей автомата, условий использования, а также требований, предъявляемых к конечному автомату.

Регистры на выходе конечного автомата могут быть установлены следующими способами:

путем использования стилей описания конечных автоматов с регистрами на выходах;

в процессе 0 при описании функций выходов в списке чувствительности указать сигнал синхронизации, т.е. вместо **always** @(*) использовать **always** (**posedge** clk);

выходной регистр описать явно в коде конечного автомата, как показано в листинге 3.

Условия переходов из каждого состояния можно описывать не только с помощью оператора **if**, но также с помощью оператора **case**. При этом обычно используется оператор **casex** с целью применения знака вопроса в константных элементах для обозначения произвольного значения бита во входном векторе.

Стиль описания надежных конечных автоматов reliability1 обеспечивает обнаружение недопустимых состояний и выполнение безусловного перехода в начальное состояние, при обнаружении недопустимого состояния, что соответствует поведению безопасного автомата (safe state machine).

Стиль описания reliability2 еще больше повышает надежность конечных автоматов. Стиль reliability2, в дополнение к стилю reliability1, для каждого состояния обеспечивает реализацию состояния ожидания в случае невыполнения ни одного из условий перехода из исходного состояния перехода.

Раздельная или векторная проверка значений входных сигналов конечных автоматов, как для автомата Мили, так и для автомата Мура, оказывает незначительное влияние на результаты синтеза и совершенно не влияет на результаты моделирования.

Раздельное присваивание значений выходных сигналов при использовании значений по умолчанию в некоторых случаях позволяет упростить логику формирования выходных сигналов и, как следствие, увеличить быстродействие конечного автомата. Кроме того, при раздельном присваивании значений выходным сигналам повышается читабельность исходного кода проекта.

Определение значения по умолчанию для состояния перехода в некоторых случаях позволяет уменьшить код проекта и увеличить быстродействие конечного автомата, однако для автоматов Мили увеличиваются временные интервалы с неопределенными значениями на выходе.

Значение по умолчанию для состояния перехода следует присваивать вначале блока **always**; не следует с этой целью использовать конструкцию **default** оператора **case**.

Определение значений по умолчанию выходных сигналов следует использовать при частом повторении одного и того же выходного вектора на различных переходах (для автомата Мили) или в различных состояниях (для автомата Мура). Это позволяет сократить исходный код проекта, а также в некоторых случаях может привести к уменьшению стоимости реализации и увеличению быстродействия.

В одном описании конечного автомата можно одновременно использовать значения по умолчанию как для состояния перехода, так и для выходных сигналов.

Из всех рассмотренных способов описания конечных автоматов для практического использования можно рекомендовать раздельное присвоение значений выходным сигналам со значениями по умолчанию и определение значения по умолчанию для состояния перехода, так как эти способы в некоторых случаях позволяют увеличить быстродействие конечного автомата, улучшить читабельность исходного кода и способствуют уменьшению стоимости реализации проекта.

В описаниях конечных автоматов важны как стили описания, так и способы описания на языках Verolog и SystemVerilog. Например, для конечного автомата Мили

из нашего примера использование способа описания, который определяет значение по умолчанию для состояния перехода, позволило увеличить быстродействие с 305 MHz до 1136 MHz, т.е. в 3,72 раза.

Литература

1. Соловьев В.В. Стили описания конечных автоматов на языке Verilog // Компоненты и технологии, 2015, № 2, с. 56-61.
2. Соловьев В.В. Логическое проектирование встраиваемых систем на FPGA. Часть 11. Стили описания конечных автоматов и кодирование внутренних состояний // Компоненты и технологии, 2019, № 8, с. 6-14.
3. Соловьев В.В. Основы языка проектирования цифровой аппаратуры Verilog. - Москва: Горячая линия - Телеком, 2021. 2-е издание, исправленное и дополненное. - 284 с. (ISBN 978-5-9912-0923-6)
4. Соловьев В.В. Язык SystemVerilog для синтеза. - Москва: Горячая линия - Телеком, 2022. 440 с. (ISBN 978-5-9912-0970-0)
5. Соловьев В.В. Язык Verilog в проектировании встраиваемых систем на FPGA. – М.: Горячая линия – Телеком. 2020. – 440 с. (ISBN 978-5-9912-0844-4)
6. Соловьев В.В. Минимизация конечных автоматов Мили путем использования значений выходных переменных для кодирования внутренних состояний // Известия Российской академии наук. Теория и системы управления, 2017, № 1, с. 89-97.
7. Salauyou V., Ostapczuk M. Finite-state State Machines Minimization by Using of Values of Input Variables at State Assignment // Measurement, Automation, Monitoring, ISSN 2450-2855, 2017, Vol. 63, nr 5, s. 195-197.
8. Соловьев В.В. Проектирование на программируемых логических интегральных схемах быстрых конечных автоматов // Проблемы разработки перспективных микро- и наноэлектронных систем - 2016. Сборник трудов / под общ. ред. академика РАН А.Л. Стемпковского. М.: ИППМ РАН, 2016. Часть I. С. 24-31.

Разница восприятия САПР QUARTUS языков SystemVerilog и VHDL и разница между VIVADO

Мальчуков А.Н.
andrey@malchukov.ru

Обсуждение и комментарии :: ссылка

Введение

Разрабатывая устройство на языках описания аппаратуры SystemVerilog или VHDL, обычно считается, что если оно описано одинаковыми конструкциями на этих языках, то и синтезируется одинаково в САПРах. На примере реализации преобразователя кода из двоичного числа в управляющие сигналы для семисегментного индикатора покажем, что это не всегда так, по крайней мере в САПР Quartus. Кроме этого, на примере одного проекта покажем, что в САПРах Quartus и Vivado по-разному работает оптимизация.

Описание проекта

Все особенности САПР, которые описываются в статье, опираются на проект-пример для студентов на плате DE10-Lite, который написан и на SystemVerilog и на VHDL. Проект-пример реализует работу с тремя байтовыми регистрами, значение которых в шестнадцатеричной системе счисления отображается на 6-ти семисегментных индикаторах – по два индикатора на каждый. С помощью двух движковых переключателей выбирается один из регистров. Значения на семисегментных индикаторах выбранного регистра начинают мигать. Новое значение устанавливается с помощью 8-ми движковых переключателей и фиксируется нажатием кнопки.

Синтез одной и той же конструкции на SystemVerilog и VHDL в разных САПР

Семисегментные индикаторы на плате DE10-Lite подключены в режиме статической индикации. Отображение символа на индикаторе с возможностью его мигания описано в отдельном модуле hex27seg, в котором преобразование 4-х разрядного двоичного числа в управляющие сигналы индикатора описано с помощью конструкции case, как на SystemVeriloge, так и на VHDL.

Преобразователь кода на SystemVerilog в модуле hex27seg:

```
always_comb begin
 case (hex)
  4'd0   : symbol_wire = 7'b1111110;
  4'd1   : symbol_wire = 7'b0110000;
  4'd2   : symbol_wire = 7'b1101101;
  4'd3   : symbol_wire = 7'b1111001;
  4'd4   : symbol_wire = 7'b0110011;
  4'd5   : symbol_wire = 7'b1011011;
  4'd6   : symbol_wire = 7'b1011111;
  4'd7   : symbol_wire = 7'b1110000;
  4'd8   : symbol_wire = 7'b1111111;
  4'd9   : symbol_wire = 7'b1111011;
  4'd10  : symbol_wire = 7'b1110111;
  4'd11  : symbol_wire = 7'b0011111;
  4'd12  : symbol_wire = 7'b1001110;
  4'd13  : symbol_wire = 7'b0111101;
  4'd14  : symbol_wire = 7'b1001111;
  4'd15  : symbol_wire = 7'b1000111;
  default: symbol_wire = 7'b0000000;
 endcase
```

Преобразователь кода на VHDL в модуле hex27seg_vhdl:

```
process (hex) begin
 case hex is
  when "0000" => symbol_wire <= "1111110";
  when "0001" => symbol_wire <= "0110000";
  when "0010" => symbol_wire <= "1101101";
  when "0011" => symbol_wire <= "1111001";
  when "0100" => symbol_wire <= "0110011";
  when "0101" => symbol_wire <= "1011011";
  when "0110" => symbol_wire <= "1011111";
  when "0111" => symbol_wire <= "1110000";
  when "1000" => symbol_wire <= "1111111";
  when "1001" => symbol_wire <= "1111011";
  when "1010" => symbol_wire <= "1110111";
  when "1011" => symbol_wire <= "0011111";
  when "1100" => symbol_wire <= "1001110";
  when "1101" => symbol_wire <= "0111101";
  when "1110" => symbol_wire <= "1001111";
  when "1111" => symbol_wire <= "1000111";
  when others => symbol_wire <= "0000000";
 end case;
end process;
```

Пройдя успешную компиляцию проектов, написанных на SystemVerilog и VHDL в САПР Quartus Prime 20.1 (далее по тексту Q20), разницы в использованных триггерах нет, а в задействованных логических ячейках разница получилась в 72 шт. между SystemVerilog и VHDL (рис. 1 и 2). В проектах модуль hex27seg подключается 6 раз, по количеству индикаторов. Получается, что модуль heg27seg в проекте на SystemVerilog на 12 лог. ячеек занимает больше, чем в проекте VHDL.

Total logic elements	286 / 49,760 (< 1 %)
Total registers	99

Рисунок 1 – результаты компиляции проекта на SystemVerilog в Q20

Total logic elements	214 / 49,760 (< 1 %)
Total registers	99

Рисунок 2 – результаты компиляции проекта на VHDL в Q20

В более старой версии САПР Quartus II 9.1 (далее по тексту Q9) наблюдается такая же картина (рис. 3 и 4).

Total logic elements	284 / 5,136 (6 %)
Total combinational functions	284 / 5,136 (6 %)
Dedicated logic registers	99 / 5,136 (2 %)

Рисунок 3 – результаты компиляции проекта на SystemVerilog в Q9

Total logic elements	212 / 5,136 (4 %)
Total combinational functions	212 / 5,136 (4 %)
Dedicated logic registers	99 / 5,136 (2 %)

Рисунок 4 – результаты компиляции проекта на VHDL в Q9

А вот САПР Vivado 2019.1 (далее по тексту V19) разницы между проектами не видит (рис. 5 и 6).

Slice LUTs (20800)	Slice Registers (41600)
141	210

Рисунок 5 – результаты компиляции проекта на SystemVerilog в V19

Slice LUTs (20800)	Slice Registers (41600)
141	210

Рисунок 6 – результаты компиляции проекта на VHDL в V19

Разница в 12 лог. ячеек между проектами на SystemVerilog и VHDL объясняется тем, что САПРы Q20 и Q9 по-разному интерпретировали конструкции case в проектах. В случае SystemVerilog – это дешифратор (рис. 7 и 8), а в случае VHDL – набор мультиплексоров (рис. 9 и 10).

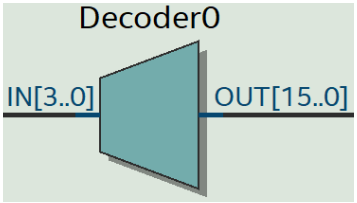


Рисунок 7 – дешифратор из RTL Viewer проекта на SystemVerilog в Q20

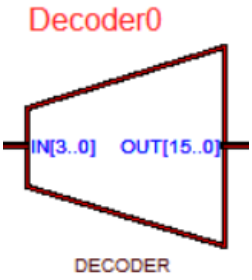


Рисунок 8 – дешифратор из RTL Viewer проекта на SystemVerilog в Q9

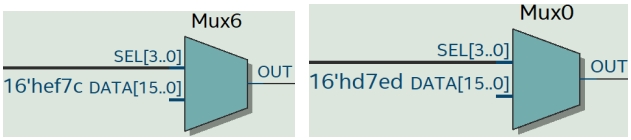


Рисунок 9 – мультиплексоры из RTL Viewer проекта на VHDL в Q20

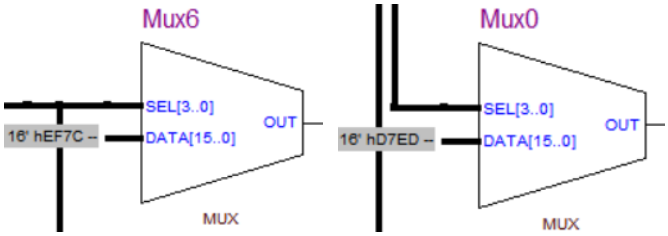


Рисунок 10 – мультиплексоры из RTL Viewer проекта на VHDL в Q9

В свою очередь в V19 в обоих проектах RTL ANALYSIS показывает одну и туже реализацию на основе RTL ROM (рис. 11).

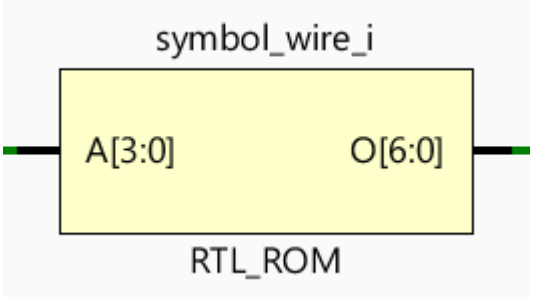


Рисунок 11 – RTL ROM из RTL ANALYSIS проекта на SystemVerilog/VHDL в V19

В подтверждении того, что разница в 72 лог. элемента между проектами на SystemVerilog и VHDL в Q20 и Q9 возникает именно из-за разного синтеза конструкций case, в проекте на SystemVerliog преобразователь был описан через мультиплексоры:

```
logic [15:0] mux6=16'hEF7C, mux5=16'hDF71,
             mux4=16'hFD45, mux3=16'h7B6D,
             mux2=16'h2FFB, mux1=16'h279F,
             mux0=16'hD7ED;
assign symbol_wire[0] = mux0[hex];
assign symbol_wire[1] = mux1[hex];
assign symbol_wire[2] = mux2[hex];
assign symbol_wire[3] = mux3[hex];
assign symbol_wire[4] = mux4[hex];
assign symbol_wire[5] = mux5[hex];
```

Результаты компиляции проекта на SystemVerilog с новым модулем hex27segMUX в Q20 (рис. 12) и Q9 (рис. 13) совпали с результатами проекта на VHDL.

Total logic elements	213 / 49,760 (< 1 %)
Total registers	99

Рисунок 12 – результаты компиляции проекта на SystemVerilog с модулем hex27segMUX в Q20

Total logic elements	212 / 5,136 (4 %)
Total combinational functions	212 / 5,136 (4 %)
Dedicated logic registers	99 / 5,136 (2 %)

Рисунок 13 – результаты компиляции проекта на SystemVerilog с модулем hex27segMUX в Q9

В V19 проект на SystemVerilog с модулем hex27segMUX по ресурсам компилируется также, как и предыдущие проекты (рис. 14).

Slice LUTs (20800)	Slice Registers (41600)
141	210

Рисунок 14 – результаты компиляции проекта на SystemVerilog с модулем hex27segMUX в V19

Исходя из описанного выше САПР Quartus по-разному реализует одинаковые устройства, описанные одинаковым образом, но на разных HDL.

Разница в оптимизации САПР Quartus и Vivado

Кроме этого, в данном проекте реализовано мигание семисегментными индикаторами, при выборе соответствующего байтового регистра движковыми переключателями. Мигание реализовано через задержку на счётчике в том же модуле hex27seg, однако байтовый регистр отображается сразу на двух индикаторах. Получается, что функция мигания задублирована, т.е. достаточно одной реализации задержки через счётчик для одновременного мигания двух индикаторов. САПР Quartus этот момент видит и лишнюю последовательностную логику из нечётных по счёту (каждый второй) модулей hex27seg вырезает (рис. 15-18), а вот САПР Vivado это не рассматривает и реализует всё как есть (рис. 19 и 20).

Entity:Instance	Logic Cells	Dedicated Logic Registers
MAX 10: 10M50DAF484C7G		
test2v2	213 (4)	99 (0)
hex27segMUX:SEG0	44 (44)	25 (25)
hex27segMUX:SEG1	14 (14)	0 (0)
hex27segMUX:SEG2	44 (44)	25 (25)
hex27segMUX:SEG3	14 (14)	0 (0)
hex27segMUX:SEG4	44 (44)	25 (25)
hex27segMUX:SEG5	14 (14)	0 (0)

Рисунок 15 – результаты компиляции проекта на SystemVerilog с расшифровкой в Q20

Entity:Instance	Logic Cells	Dedicated Logic Registers
MAX 10: 10M50DAF484C7G		
test2v2_vhdl	214 (4)	99 (0)
hex27seg_vhdl:SEG0	45 (45)	25 (25)
hex27seg_vhdl:SEG1	14 (14)	0 (0)
hex27seg_vhdl:SEG2	44 (44)	25 (25)
hex27seg_vhdl:SEG3	14 (14)	0 (0)
hex27seg_vhdl:SEG4	44 (44)	25 (25)
hex27seg_vhdl:SEG5	14 (14)	0 (0)

Рисунок 16 – результаты компиляции проекта на VHDL с расшифровкой в Q20

Entity	Logic Cells	Dedicated Logic Registers
Cyclone III: AUTO		
test2v2	212 (3)	99 (0)
hex27segMUX:...	44 (44)	24 (24)
hex27segMUX:...	15 (15)	1 (1)
hex27segMUX:...	44 (44)	24 (24)
hex27segMUX:...	15 (15)	1 (1)
hex27segMUX:...	44 (44)	24 (24)
hex27segMUX:...	15 (15)	1 (1)

Рисунок 17 – результаты компиляции проекта на SystemVerilog с расшифровкой в Q9

Entity	Logic Cells	Dedicated Logic Registers
Cyclone III: AUTO		
test2v2_vhdl	212 (3)	99 (0)
hex27seg_vhdl:...	44 (44)	24 (24)
hex27seg_vhdl:...	15 (15)	1 (1)
hex27seg_vhdl:...	44 (44)	24 (24)
hex27seg_vhdl:...	15 (15)	1 (1)
hex27seg_vhdl:...	44 (44)	24 (24)
hex27seg_vhdl:...	15 (15)	1 (1)

Рисунок 18 – результаты компиляции проекта на VHDL с расшифровкой в Q9

Name	Slice LUTs (20800)	Slice Registers (41600)
test2v2	141	210
SEG0 (hex27segMUX)	11	31
SEG1 (hex27segMUX_0)	12	31
SEG2 (hex27segMUX_1)	11	31
SEG3 (hex27segMUX_2)	12	31
SEG4 (hex27segMUX_3)	11	31
SEG5 (hex27segMUX_4)	12	31

Рисунок 19 – результаты компиляции проекта на SystemVerilog с расшифровкой в V19

Name	Slice LUTs (20800)	Slice Registers (41600)
test2v2_vhdl	141	210
SEG0 (hex27seg_vhdl)	11	31
SEG1 (hex27seg_vhdl_0)	12	31
SEG2 (hex27seg_vhdl_1)	11	31
SEG3 (hex27seg_vhdl_2)	12	31
SEG4 (hex27seg_vhdl_3)	11	31
SEG5 (hex27seg_vhdl_4)	12	31

Рисунок 20 – результаты компиляции проекта на VHDL с расшифровкой в V19

Для САПР Vivado надо самому просчитывать дублирующие моменты заранее и избегать их. Зато, в отличие от САПР Quartus, в САПР Vivado в VHDL поддерживаются многовходовые логические элементы в нотации 2008:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity logic_gates_vhdl is
port (a, b: in std_logic;
    f_and, f_nand, f_or, f_nor: out std_logic;
    f_xor, f_xnor, f_not_a: out std_logic);
end logic_gates_vhdl;
architecture Behavioral of logic_gates_vhdl is
begin
    f_not_a <= not a;
    f_and <= a and b;
    f_nand <= a nand b;
    f_or <= a or b;
    f_nor <= a nor b;
    f_xor <= a xor b;
    f_xnor <= a xnor b;
end Behavioral;
```

Заключение

САПР Quartus может по-разному синтезировать одинаковые устройства, описанные одинаковыми конструкциями на SystemVerilog и VHDL, в отличие от САПР Vivado. Кроме того, Quartus может обнаруживать и удалять лишние дублирующие конструкции, тем самым уменьшая затрачиваемые ресурсы, а Vivado, судя по всему, этим не занимается. Зато Vivado в VHDL поддерживаются многовходовые логические элементы в нотации 2008, в отличие от Quartus.

УДК 004.021

Оптимизация вычислительных структур под архитектуру ПЛИС XILINX

Алексеев К.Н., *к.т.н*
ООО «Радио Гигабит», г. Нижний Новгород, Россия
e-mail: alexseev91@mail.ru

Сорокин Д.А., *к.т.н*
ООО «НИЦ СЭ и НК, г. Таганрог, Россия
jotun@inbox.ru

Обсуждение и комментарии :: ссылка

Аннотация

Задача оптимизации вычислительных структур под архитектуру конкретной ПЛИС относится к классу задач многокритериальной оптимизации, решение которой позволяет обеспечить целевые показатели производительности системы. Необходимость оптимизации особо остро возникает при работе с высоконагруженными проектами, для которых характерны: высокая утилизация ПЛИС – более 50%; использование большого числа примитивов встроенной памяти (BRAM, URAM) и встроенных арифметических блоков (DSP); высокие целевые тактовые частоты. На практике, для решения задачи оптимизации обычно предпринимают комплекс мер, обеспечивающих требуемые целевые характеристики вычислительной структуры.

Методологии проектирования, разработанные вендорами ПЛИС, предлагают оптимальную последовательность этапов разработки проекта, на каждом из которых предлагается использовать набор лучших практик. Однако, как правило в документации информация представляется в сжатом виде и рассматриваются только наиболее важные аспекты проектирования вместе с инструментами разработки и автоматизации. В связи с этим, тема оптимизации вычислительных структур освещается лишь косвенно: информация по ней не систематизирована и не всегда должным образом проиллюстрирована.

В данной работе предложен набор методов оптимизации вычислительных структур, дополняющий лучшими практиками общую методологию проектирования. Проиллюстрировано использование предложенных методов на конкретных типовых примерах.

Применение предложенных методов способствовало некоторому увеличению степени масштабирования вычислений и достижению высоких тактовых частот работы вычислительных структур ряда прикладных вычислительно-трудоемких задач разных предметных областей, за счет чего результирующая производительность системы была увеличена на 10% – 50%.

Ключевые слова: ПЛИС; FPGA; САПР; CAD; Physical Constraints; Placement Constraints; Timing Closure.

Введение

Программируемые логические интегральные схемы (ПЛИС) успешно применяются в гибридных и реконфигурируемых вычислительных системах (ГВС и РВС) [1] для ускорения специализированных вычислений. Реализация в базисе ПЛИС прикладных вычислительно-трудоемких задач разных предметных областей позволяет получить более высокую реальную производительность системы за счет конвейеризации вычислительного процесса, не смотря на сравнительно низкую тактовую частоту работы схемы [2-7]. Для достижения наибольших показателей производительности системы, необходимо использовать максимум аппаратного ресурса, обеспечив наибольшую степень масштабирования на целевой тактовой частоте [8].

При реализации высоконагруженных проектов, объединение разработанных и отлаженных блоков в едином вычислительном контуре зачастую не позволяет достигать высоких тактовых частот работы схемы, что происходит вследствие ограниченности возможностей алгоритмов САПР: размещения элементов на вычислительном поле ПЛИС и трассировки соединений между используемыми примитивами (place and route). Увеличение утилизации ПЛИС приводит к увеличению числа используемых коммутационных трасс, из-за чего сокращается количество возможных благоприятных вариантов трассировки и как следствие увеличивается длина прокладываемых связей. Зачастую алгоритмы place and route в автоматическом режиме не могут удовлетворить требованиям к целевой тактовой частоте,

не смотря на широкие возможности САПР в автоматизированной оптимизации вычислительной структуры, направленной на уменьшение длины трасс между примитивами. В результате для достижения требуемых характеристик производительности перед разработчиками остро встает необходимость оптимизации вычислительной структуры, для чего необходимо понимание архитектурных возможностей целевой ПЛИС и методики организации эффективных параллельно-конвейерных вычислений [8].

Существует множество методов проектирования, представляющих собой свод рекомендаций по разработке эффективных вычислительных структур [9, 10]. Следование обозначенным правилам позволяет в результате достигать требуемой производительности системы. Вместе с этим, не все методы оптимизации, эффективно применяемые на практике, широко освещены в литературе, а при описании конкретных инструментов не всегда присутствуют наглядные примеры по их применению.

Целью данной работы является описание методов оптимизации вычислительной структуры реальных прикладных задач под архитектуру ПЛИС Xilinx. Задача оптимизации является многокритериальной, поэтому работе будут освещено несколько различных подходов или сторон рассматриваемой проблемы. Описанные методы призваны в некотором роде расширить распространенные методики проектирования при решении комплексных, вычислительно-трудоемких задач в базисе ПЛИС.

Методы оптимизации вычислительной структуры.

Рассмотрим несколько методов оптимизации вычислительных структур, направленных на использование архитектурных особенностей целевой ПЛИС. Часть методов предполагает не только внесение изменений в разработанную вычислительную структуру, но и переработку подсистемы синхронизации; другая часть направлена на оптимизацию работы алгоритмов размещения без необходимости изменения вычислительной структуры. Все описанные методы апробированы при решении прикладных вычислительно-трудоемких задач в базисе ПЛИС.

1. Оптимизация связей с высокой степенью ветвления (fanout).

Если источник сигнала имеет связь всего с одним приемником, алгоритмы САПР стараются обеспечить оптимальное размещение примитивов относительно друг друга и удовлетворить ограничениям по времени распространения сигнала, которые определены тактовой частотой. Чем больше степень ветвления сигнала, тем сложнее оптимально разместить множество взаимосвязанных элементов на поле примитивов ПЛИС и проложить между ними трассы требуемой длины.

На рисунке 1-а приведена иллюстрация схемы с одним источником сигнала S и множеством приемников (регистров) R_1-R_n. Обычно подобная ситуация наблюдается на шине загрузки/выгрузки данных или при использовании сигналов управления: clock enable и set/reset. Уменьшить степень ветвления в общем случае можно размножив источник сигнала, например, как это показано на рисунке 1-б. Этот метод автоматизирован в САПР Xilinx Vivado, для его использования необходимо применять HDL attribute или TCL команду max_fanout [9-11]. Однако, исследования показали, что использование данной опции, во-первых, не гарантирует получения оптимального количества размноженных источников S, а во-вторых, часть источников может иметь в разы больше связей с приемниками сигнала (рисунок 1-в). В этом случае рационально либо выполнять более тонкую настройку параметров инструмента max_fanout, по возможности ограничив число связей размножаемых источников, либо вручную оптимизировать вычислительную структуру.

Вместе с этим, когда дело касается размножения именно шины данных, существует более эффективное средство борьбы с fanout: организация работы с общей шиной, схема которой показана на рисунке 1-г. Конвейеризация процесса загрузки/выгрузки данных позволяет значительно уменьшить число ветвлений при пропорциональном увеличении числа синхронизирующих элементов. Использовать схему с общей шиной возможно в случае, когда данные поступают от одного источника, например, из одного интерфейсного модуля или примитива памяти.

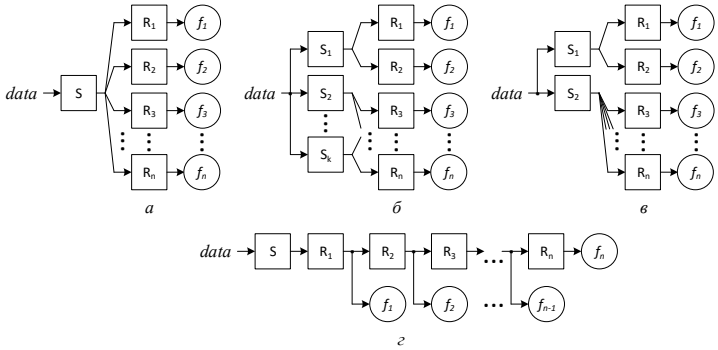


Рисунок 1 – Схемы передачи сигнала: а) от одного источника S множеству приемников R; б) от множества источников S множеству приемников R; в) от двух источников S множеству приемников R; г) по общей шине

Стоит отметить, что в результате использования общей шины меняется синхронизация функциональных блоков относительно друг друга. Разработчик должен учитывать изменение поведения вычислительной структуры в алгоритме задачи путем компенсации задержки между загружаемыми данными в том числе и в подсистеме управления вычислительным процессом.

2. Оптимизация числа управляющих сигналов триггеров.

В ПЛИС фирмы Xilinx семейств UltraScale и UltraScale+ основной структурной единичей логического ресурса являются так называемые CLB (Configurable logic blocks), каждый из которых содержит 8 LUT (Look Up Table) и 16 FF. При этом сигналы управления (control set) на входах триггеров: clock enable (CE) и set/reset, объединены в группы и одновременно заведены сразу на 4 FF. Таким образом, в каждый CLB поступает всего 4 выделенные линии управляющих сигналов [12]. В связи с этим, в высоконагруженных проектах с утилизацией ресурсов более 50% и большим числом FF с уникальным набором сигналов управления можно наблюдать ситуацию, когда в CLB использовано всего несколько триггеров. В этом случае становится сложно удовлетворить требованиям тактовой частоты для трасс всего проекта.

Алгоритмы Vivado позволяют автоматически уменьшить число управляющих сигналов с помощью использования впередистоящих LUT, как это показано на рисунке 2. Данную опцию можно включить на этапе синтеза control_set_opt_threshold или с помощью команды TCL control_set_remap [9-11].

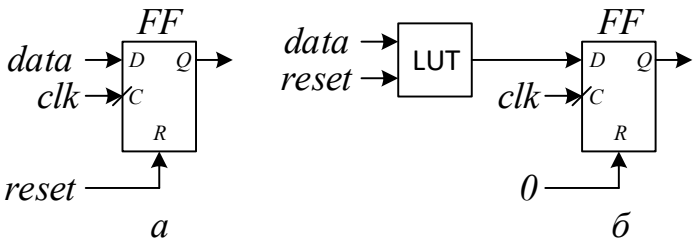


Рисунок 2 – Триггер FF: а) сигнал reset поступает на вход R; б) сигнал reset идет через LUT на вход D

Однако, очевидно, что наилучшим вариантом является отказ от использования большого числа сигналов управления в реализуемой вычислительной структуре. Для определения такой возможности, стоит руководствоваться следующим набором правил:

1) не использовать reset для сброса шин данных в конвейере. Зачастую необходимость очистки конвейера от данных возникает только в начале или в конце работы алгоритма, в связи с чем обычно достаточно подать сигнал сброса необходимой длины лишь на последний регистр. Если же работа обработка данных ведется по маркеру, достаточно подать сброс лишь на логику управления (рисунок 3-а). В этом случае конвейер будет обрабатывать как полезные, так и неактуальные, «мусорные» данные, однако только нужный результат будет валидирован маркером.

2) если необходимо выполнить сброс данных, поступающих с примитивов памяти BRAM (рисунок 3-б), можно сбросить именно выходной регистр самого примитива [13]. Это позволит не использовать сигналы управления FF, и уменьшит число трасс для сброса до одной, поступающей на вход примитива BRAM.

3) примитивы DSP ПЛИС Xilinx имеют встроенные регистры с уникальными наборами управляющих сигналов [14], поэтому рационально их использовать для хранения входных операндов-констант (рисунок 3-в). Такой подход вместе с числом используемых трасс позволит также сократить и количество триггеров FF, используемых ранее для хранения операндов.

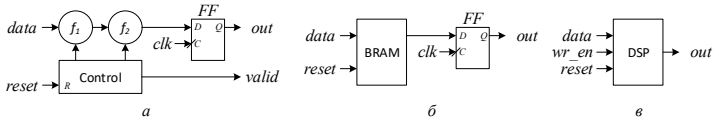


Рисунок 3 – а) сигнал reset сбрасывает логику управления, а не регистры конвейера; б) сигнал reset сбрасывает выходной регистр памяти BRAM, а не внешний регистр FF; в) по сигналу wr_en данные data записываются во внутренний регистр DSP, а сигнал reset очищает этот регистр

3. Оптимизация работы с примитивами BRAM и URAM.

Практика показала, что использование в проекте более 50% имеющегося ресурса BRAM значительно ухудшает показатели достижимой тактовой частоты. В первую очередь это связано с задержкой сигнала на самом примитиве. Получить хорошую производительность в проекте с высокой утилизацией BRAM возможно, если использовать встроенный в примитив выходной регистр данных (рисунок 4) [13]. В свойствах (properties) примитива BRAM использование выходного регистра обозначено как DOA_REG / DOB_REG = 1. Стоит учитывать, что задержка на чтение данных (read latency) будет равна двум, что не всегда может быть рационально для используемого алгоритма обработки данных, особенно при наличии обратных связей.

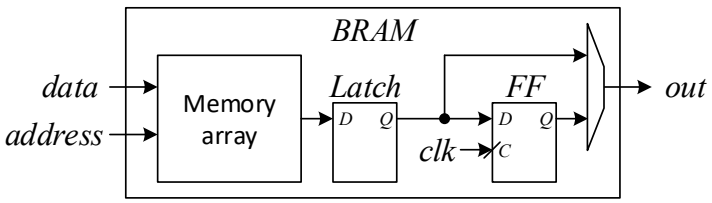


Рисунок 4 – Упрощенная схема примитива блочной памяти BRAM

Также примитив BRAM имеет настройку выходного порта, описывающую поведение при одновременном чтении и записи в одну и ту же ячейку памяти: READ_FIRST, WRITE_FIRST и NO_CHANGE. Документация Xilinx [15] регламентирует максимальную частоту работы примитива в каждом из режимов в зависимости от спидгрейда ПЛИС. Режим работы примитива BRAM напрямую влияет на характеристику временного анализа Pulse Width. Наибольшую тактовую частоту обеспечивает режим WRITE_FIRST, поэтому при возможности рекомендуется использовать именно его, особенно если работа идет с ПЛИС со спидгрейдом -1.

Примитив URAM имеет три дополнительных конфигурируемых встроенных регистра: один регистр входных данных, и два – выходных данных. Использование данных регистров также позволяет оптимизировать задержку выходного сигнала с примитива и пропорционально увеличивает параметр read latency.

Одни и те же функциональные узлы вычислительной структуры могут быть реализованы с использованием различных примитивов ПЛИС, например, хранение небольших массивов данных можно организовать с использованием: регистров; распределенной памяти; примитивов BRAM; примитивов URAM. В связи с этим можно использовать метод балансировки аппаратного ресурса, предлагающий в общем случае уменьшать число LUT и FF за счет максимального использования доступных примитивов DSP, BRAM и URAM. Подробнее принципы балансировки ресурсов и их применение описаны в работе [16]. Помимо прочего, балансировка аппаратного ресурса в некоторых случаях может позволить несколько увеличить степень параллелизма.

4. Оптимизация размещения вычислительной структуры в ПЛИС.

Основной операцией САПР, которая оказывает наибольшее влияние на результирующую тактовую частоту является размещение синтезированной вычислительной структуры в примитивах ПЛИС. При высокой утилизации ресурсов ПЛИС именно алгоритмы размещения позволяют удовлетворить требованиям к длине трасс между примитивами, что в большей степени и определяет предельную тактовую частоту вычислительной структуры.

Исследования показывают [8], что алгоритмы размещения не способны обеспечить оптимальную установку всех элементов вычислительной структуры относительно друг друга. Элементы соседних блоков начинают конкурировать за использование наиболее коротких трасс между ними, из-за чего возникает излишняя нагрузка на коммутационную матрицу ПЛИС, и как результат может быть обеспечена работа на меньшей таковой частоте. Ситуация усугубляется с увеличением степени утилизации примитивов DSP, BRAM и URAM из-за их неоптимального размещения средствами САПР в автоматическом режиме.

Также исследования показали, что при высокой степени утилизации ПЛИС, на результирующую тактовую частоту оказывают большое влияние связи между примитивами, которые расположены в областях ПЛИС, физически разделенных встроенными переферийными устройствами, такими как контроллеры памяти, Ethernet и каналы ввода/вывода Pins (рисунок 5). Вместе с этим, наиболее длинные связи возникают при пересечении границ SLR (superlogic region) – отдельных кремниевых подложек, объединенных в корпусе микросхемы [17].

Рисунок 5 – Трасса, пересекающая области ввода/вывода Pins

Для достижения целевой тактовой частоты была предложена методика создания топологических ограничений ПЛИС [8], направленная на минимизацию длин трасс между примитивами за счет рационального размещения элементов функциональных узлов в заданной топологической области поля примитивов. Методика предполагает выполнение нижеописанной совокупности действий:

1) разработать траекторию размещения узлов вычислительной структуры, которая обеспечит размещение в непосредственной близости узлов вычислительной структуры, имеющих функциональную зависимость по приему и передаче данных между собой и зависимости от узлов управления ходом вычислений. Траектория размещения должна учитывать местоположение внешних периферийных устройств и обеспечивать минимум пересечений трасс топологических областей ПЛИС, содержащих аппаратно-реализованные периферийные устройства и границы SLR.

2) разместить примитивы DSP, BRAM и URAM в соответствии с разработанной траекторией. Размещение примитивов выполняется путем создания системы топологических ограничений инструментами Xilinx Physical Constraints: PBlock (Physical Block) – физическое ограничение отдельной области ПЛИС для реализации в ней некоторых функциональных узлов; LOC – размещение элемента узла в примитиве ПЛИС с указанными топологическими координатами. Практика показала, что рациональное размещение примитивов DSP, BRAM и URAM в соответствии с разработанной траекторией зачастую позволяет обеспечить достижение целевой тактовой частоты.

3) с помощью инструмента PBlock уменьшить длины трасс, локализовав примитивы LUT и FF в требуемой топологической области ПЛИС, например, в непосредственной близости от зависимых примитивов DSP, BRAM и URAM.

4) с помощью инструмента PBlock минимизировать число трасс между примитивами, расположенными на разных SLR или разделенными аппаратно-реализованными периферийными устройствами. Инструмент user_crossing_SLR для локализации связей в одном конкретном SLR [9-11].

При размещении узлов вычислительной структуры в PBlock, необходимо избегать ситуации, когда утилизация ресурсов различных топологических областей будет существенно отличаться. В этом случае возникнет неравномерная нагрузка на коммутационную матрицу ПЛИС, что негативно сказывается на результирующей тактовой частоте. Избежать данной ситуации можно разместив один крупный функциональный узел вычислительной структуры в разных топологических областях ПЛИС, по возможности, обеспечив связи с нулевой логической нагрузкой между областями (связи типа «триггер-триггер»: logic levels = 0). Также для достижения лучшей степени параллелизма можно выполнить балансировку ресурса вычислительной структуры, изменив число используемых примитивов DSP, BRAM, URAM [16].

5) если это возможно, внести изменения в вычислительную структуру, которые бы обеспечили нулевую логическую нагрузку между примитивами, расположенными в разных SLR.

5. Оптимизация трасс путем ограничений временного анализа

Ограничения при выполнении временного анализа с помощью инструментов timing constraints позволяют определять поведение как сигналов, состояние которых меняется с иным периодом целевой тактовой частоты, так и полностью асинхронных сигналов. Поведение и области применения основных инструментов timing constraints: set_multicycle_path, set_false_path и set_max_delay, описаны в документации Xilinx [9-11], однако подробно разобраны только случаи их применения для согласования переходов сигнала между тактовыми доменами. Оптимизация работы алгоритмов размещения и трассировки с помощью данных инструментов используется достаточно редко, так как их неосторожное применение может привести к возникновению явления метастабильности на выходе триггеров [9, 11, 18], что влечет за собой риски, связанные полной неработоспособностью прошивки ПЛИС. Однако, при рациональном и продуманном использовании, timing constraints становятся мощным инструментом оптимизации, направленном на достижение целевых тактовых частот в высоконагруженных проектах.

1) Согласно документации [11], инструмент set_multicycle_path позволяет задавать временные ограничения для сигналов, состояние которых необходимо отслеживать, но частота изменений заведомо больше такта целевой частоты. Результатом использования инструмента set_multicycle_path будет трасса, удовлетворяющая конкретным параметрам setup и hold, измеряемых в тактах рабочей частоты и имеющая пропорционально длинный путь. Использование данного инструмента обосновано, например, когда заранее известно, что данные, записанные в конкретный регистр, будут использованы не в следующем такте, а только через определенное время. Применение set_multicycle_path помогает алгоритмам place and route сосредоточиться на трассировке иных путей, к которым предъявляются более жесткие требования тактовой частоты.

2) Инструмент set_false_path [11] позволяет полностью исключить конкретные связи из временного анализа. При его использовании САПР Vivado трассирует соединения между примитивами без анализа длины пути так, чтоб они не мешали трассам других сигналов. По рекомендациям Xilinx данный инструмент рационально использовать для оптимизация глобального сброса, приводящего схему к исходному состоянию.

Данный вариант оптимизации возможен только в том случае, если после глобального сброса работа схемы начнется со значительной по времени задержкой. В отличие от set_multicycle_path, который фактически определяет максимальную длину пути, set_false_path не привязан к конкретным параметрам setup и hold, что теоретически может вызвать метастабильность на

выходе триггеров. Однако, при сравнительно долгом удержании сигнала глобального сброса и большом времени ожидания запуска дальнейших вычислений, риск некорректной работы стремится к нулю.

Вместе с этим, представители фирмы Xilinx рекомендуют сигнал глобального сброса или установки пускать по трассам тактовых сигналов. Это можно сделать только в случае, когда сигнал сброса не проходит через LUT, а поступает на соответствующий вход FF. Для перевода сигнала с обычных трасс на тактовые линии необходимо использовать примитивы BUFG [17].

Исследования показали, что применение ограничения set_false_path к глобальному сбросу и его перевод на тактовые трассы позволяет значительно облегчить этап обеспечения временных параметров иных сигналов. Это происходит вследствие того, что источник сигнала сброса имеет зачастую огромное число ветвлений и упрощение требований к его трассировке очевидно дает большее число вариантов трасс для иных сигналов. Отметим, что в высоконагруженном проекте, выполненном на ПЛИС серии UltraScale со спидгрейдом -1 и целевой частоте 250 МГц, при утилизации ресурсов около 70% максимальная длина пути сигнала глобального сброса с числом ветвлений около 2000 и ограничением set_false_path достигла порядка 40ns.

3) Помимо оптимизации сигнала глобального сброса инструмент set_false_path можно использовать для ограничений трассировки путей к вычислительным блокам от триггеров, хранящих загружаемые единожды настроечные константы. На практике можно накладывать ограничения не только на однократно загружаемые константы, но и на перегружаемые в процессе работы. Если для загрузки и хранения констант используется общая шина, можно наложить ограничение set_false_path от регистра к приемнику сигнала, не нарушив при этом синхронизацию цепи регистров (рисунок 6).

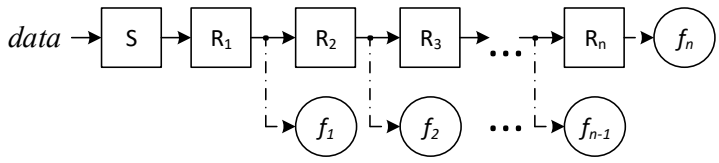


Рисунок 6 – Схема с общей шиной: штрих-пунктирной линией показаны сигналы с ограничением set_false_path

Отметим, что использование методов 2) и 3) оптимизации временных ограничений трасс позволяет также несколько сократить ресурс FF, используемый для синхронизации глобального сброса или схемы передачи констант от интерфейсного блока на большие расстояния.

Апробация

Применение рассмотренных методов оптимизации апробировано при решении задачи нагрузочного тестирования ПЛИС. В работе [8] проиллюстрировано использование методики создания топологических ограничений, которая позволила обеспечила прирост производительности на 20% за счет работы вычислительной структуры в ПЛИС Xilinx Kintex UltraScale XCKU095 на целевой тактовой частоте 500 МГц. На рисунке 7 представлено визуальное отображение размещения узлов вычислительной структуры средствами САПР Vivado 2020.2. Очевидно, что узлы расположены в неоптимальном порядке, что дает значительную дополнительную нагрузку на коммутационную матрицу ПЛИС.

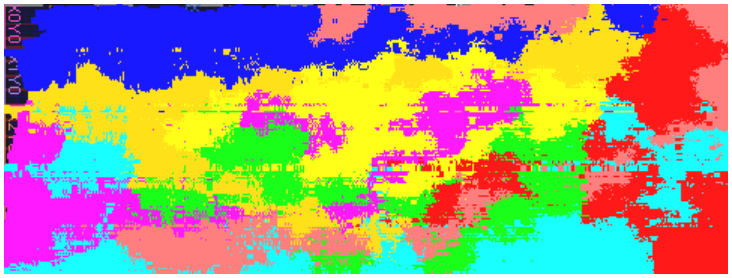


Рисунок 7 – Размещение узлов вычислительной структуры задачи нагрузочного тестирования в ПЛИС XCKU095

На рисунке 8 представлена траектория размещения узлов вычислительной структуры, соответствующая расположению столбцов DSP блоков. Черные точки иллюстрируют границы переходов между областями ввода/вывода Pins, а серые круги отражают расположение интерфейсов, по которым осуществляется загрузка и выгрузка конвейера.

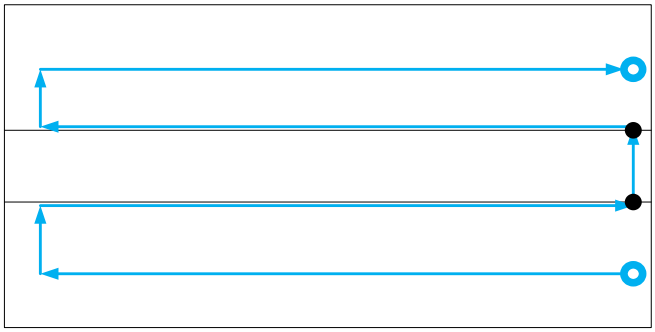


Рисунок 8 – Траектория размещение узлов вычислительной структуры задачи нагрузочного тестирования ПЛИС XCKU095

Визуализация размещения узлов вычислительной структуры после применения методики создания топологических ограничений (рисунок 9) косвенно свидетельствует об оптимизации нагрузки на коммутационную матрицу ПЛИС.

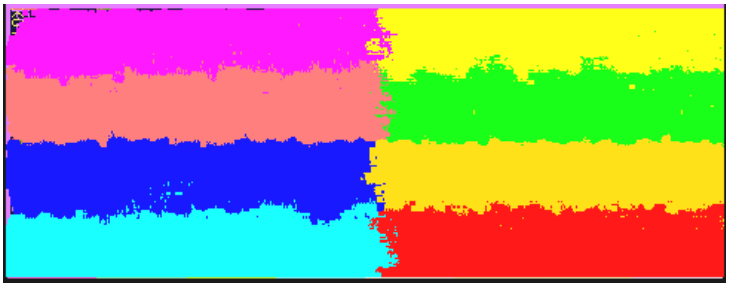


Рисунок 9 – Результат размещения узлов вычислительной структуры задачи нагрузочного тестирования после использования инструментов LOC

В работе [16] показано, что использование балансировки конфигурации узлов вычислительной структуры задачи нагрузочного тестирования ПЛИС Xilinx Virtex UltraScale+ VU9P позволило увеличить степень параллелизма на 9% за счет использования всех доступных примитивов DSP. При этом, использование методики создания топологических ограничений позволило обеспечить прирост производительности примерно на 30% за счет работы вычислительной структуры на целевой тактовой частоте 525 МГц. На рисунке 10 представлена траектория размещения узлов вычислительной структуры, соответствующая расположению столбцов DSP блоков, в которой также минимизировано число переходов между SLR. Серые круги отражают расположение интерфейсов, по которым осуществляется загрузка и выгрузка конвейера, а стрелки, выделенные разным штрихом иллюстрируют различную конфигурацию узлов вычислительной структуры. Так, каждый узел нижней области использует 14 примитивов DSP, а узел в верхней области – 21 примитив DSP.

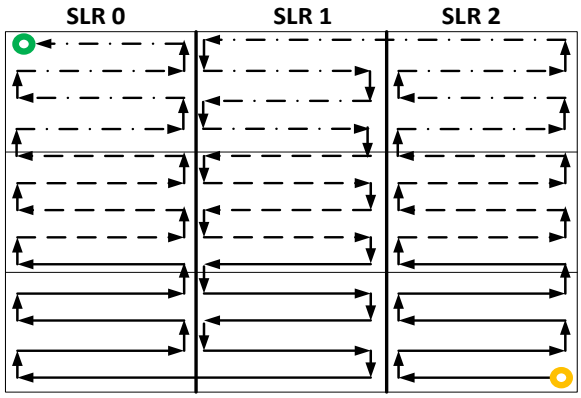


Рисунок 10 – Траектория размещение узлов вычислительной структуры задачи нагрузочного тестирования ПЛИС XCKU095

Вместе с этим применение рассмотренных методов позволило увеличить производительность системы при решении иных вычислительно-трудоемких задач [2-7], например, при решении СЛАУ методом LU разложения, производительность РВС, построенной на базе ПЛИС Xilinx Kintex UltraScale XCKU095, была увеличена более чем на 20% за счет достижения целевой тактовой частоты 435 МГц [5].

Заключение.

В рамках данной работы рассмотрены методы оптимизации вычислительных структур, направленные на достижение целевых тактовых частот. Задача оптимизации характеризуется поиском компромиссного решения относительно следующих критериев: степень утилизации основного логического ресурса LUT (максимизация); степень утилизации встроенных аппаратно-реализованных арифметических блоков и блоков памяти (максимизация); общее число используемых примитивов ПЛИС (минимизация); длина связей между примитивами ПЛИС (минимизация); число связей между физически разделенными топологическими областями ПЛИС (минимизация); частота работы вычислительной структуры (максимизация).

Рассмотренные методы оптимизации вычислительных структур актуальны при решении на ПЛИС прикладных задач разных предметных областей. Автоматизация предложенных методов теоретически позволит значительно сократить время отладки вычислительных структур, требуемое для достижения целевой производительности системы.

Библиографический список

1. Каляев И.А., Левин И.И., Семерников Е.А., Шмойлов В.И. Реконфигурируемые мультиконвейерные вычислительные структуры. Изд. 2-е, перераб. и доп. / Под общ. ред. И.А. Каляева. - Ростов н/Д: Издательство ЮНЦ РАН, 2009. – 344 с. ISBN 978-5-902982-61-6.

2. Alekseev K., Levin I., Sorokin D. Implementation of surface-related multiple prediction task on reconfigurable computer systems // Bulletin of the South Ural State University, Series: Mathematical Modelling, Programming and Computer Software, 2020, 13 (1), pp. 81-94.

3. Алексеев К.Н., Сорокин Д.А., Матросов А.Ю., Семерникова Е.Е. Структурно-процедурная реализация алгоритма прогнозирования кратных волн на ПЛИС // Известия ЮФУ. Технические науки. – Ростов/Д.: Изд-во ЮФУ, 2016. – № 12. – С. 16-28

4. Алексеев К.Н., Левин И.И. Реализация обратной кинематической задачи сейсморазведки для микросейсмического мониторинга на реконфигурируемых вычислительных системах в реальном масштабе времени // Известия ЮФУ. Технические науки. – Ростов/Д: Изд-во ЮФУ, 2018. –

№8 (202). - С. 221-231.

5. Левин И.И. Пелипец А.В. Эффективная реализация распараллеливания на реконфигурируемых системах // Вестник компьютерных и информационных технологий. – М.: Машиностроение, 2018. – № 8. –С. 11–16.
6. Левин И.И., Доронченко Ю.И., Сорокин Д.А., Чистяков А.Е. Моделирование распространения акустических волн в массивной породе с применением реконфигурируемой вычислительной системы // Нефтяное хозяйство. -М.: ЗАО «Издательство «Нефтяное хозяйство», 2016. -№3. -С.50-53.
7. Сорокин, Д.А., Дордопуло А.И. Методика сокращения аппаратных затрат в сложных системах при решении задач с существенно-переменной интенсивностью потоков данных // Известия ЮФУ. Технические науки. – Таганрог: Изд-во ТТИ ЮФУ, 2012. – №4. – С. 213-219.
8. Алексеев К.Н., Сорокин Д.А., Леонтьев А.Л. Метод управления размещением элементов вычислительной структуры при максимальной утилизации ресурсов ПЛИС // XIV Всероссийская мультиконференция по проблемам управления (МКПУ-2021): материалы XIV мультиконференции (Дивноморское, Геленджик, 27 сентября – 2 октября 2021 г.): в 4 т. / Южный федеральный университет [редкол.: И.А. Каляев, В.Г. Пешехонов и др.]. – Ростов-на-Дону; Таганрог: Издательство Южного федерального университета, 2021. ISBN 978-5-9275-3846-1 Т. 2: – 284 с. - С. 238-240.
9. UltraFast Design Methodology Guide for FPGAs and SoCs (UG949) [электронный ресурс]. – Режим доступа: https://docs.xilinx.com/r/en-US/ug949-vivado-design-methodology (дата обращения: 04.11.2023)
10. Vivado Design Suite User Guide: Design Analysis and Closure Techniques [электронный ресурс]. – Режим доступа: https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2021_2/ug906-vivado-design-analysis.pdf#nameddest=xPerformingTimingAnalysis (дата обращения: 04.11.2023)
11. Vivado Design Suite User Guide: Using Constraints [электронный ресурс]. – Режим доступа: https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2021_1/ug903-vivado-using-constraints.pdf (Дата обращения: 04.11.2023)
12. UltraScale Architecture Configurable Logic Block [электронный ресурс]. – Режим доступа: https://docs.xilinx.com/v/u/en-US/ug574-ultrascale-clb (дата обращения: 04.11.2023)
13. UltraScale Architecture Memory Resources User Guide [электронный ресурс]. – Режим доступа: https://docs.xilinx.com/v/u/en-US/ug573-ultrascale-memory-resources (дата обращения: 04.11.2023)
14. UltraScale Architecture DSP Slice User Guide (UG579) [электронный ресурс]. – Режим доступа: https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp (дата обращения: 04.11.2023)
15. Kintex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics (DS892) [электронный ресурс]. – Режим доступа: https://docs.xilinx.com/v/u/en-US/ds892-kintex-ultrascale-data-sheet (дата обращения: 04.11.2023)
16. Алексеев К.Н., Сорокин Д.А. Управление процессом синтеза вычислительной структуры при решении вычислительно-трудоемких задач на ПЛИС // Труды 16 -ой Всероссийской мультиконференции по проблемам управления (МКПУ-2023) 11-15 сентября 2023 г., г. Волгоград, Россия. – Волгоград. ВолгГТУ, 2023. Т.2. – С. 128-133. ISBN 978-5-9948-4705-3 (Т. 2)
17. UltraScale Architecture and Product Data Sheet: Overview [электронный ресурс]. – Режим доступа: https://docs.xilinx.com/v/u/en-US/ds890-ultrascale-overview (дата обращения: 04.11.2023)
18. Строгонов А. Неизвестное об известном, или что такое метастабильность триггеров //Компоненты и технологии. – 2008. – №. 87. – С. 141-144.

Реализация видеовывода сверхвысокой четкости на микросхемах семейства Zynq-7000

М. А. Попов,
гл. спец. по программному обеспечению
ООО «Бигпринтер Цифровые Инновации»
e-mail: maks@bigprinter.ru

А. Ю. Романов,
канд. техн. наук, доц. Национальный исследовательский университет «Высшая школа экономики»
e-mail: a.romanov@hse.ru

Обсуждение и комментарии :: ссылка

Аннотация

Статья описывает способ построения подсистемы вывода изображения сверхвысокого разрешения (2560 x 1440 пикселей и выше), методику оценки допустимых параметров видеорежима, а также пример успешной реализации такой подсистемы на микросхемах начального уровня семейства *Zynq-7000*. Приведены результаты испытаний с мониторами различного разрешения.

Ключевые слова: *Zynq*, высокое разрешение, видеовывод, *Petalinux*, пользовательский интерфейс

Введение

Развитый пользовательский интерфейс (ПИ) является важнейшей частью многих современных встраиваемых систем. Не в последнюю очередь возможности ПИ определяются параметрами видеоизображения, которое способен воспроизвести монитор системы. Если еще 5 лет назад графический режим *FullHD* (1920 x 1080 точек) удовлетворял практически любые требования пользователей, то сейчас не являются редкостью режимы *2K* (2560 x 1440 пикселей), *4K* (3840 x 2160 пикселей) и даже выше. Настолько большое количество пикселей в кадре резко повышает требования ко всем составляющим встраиваемой системы, начиная от производительности процессорного ядра (ядер) и заканчивая качеством трассировки и изготовления печатной платы (ПП).

Такие требования подразумевают использование и современной элементной базы (например, *MPSoC* семейства *Xilinx Zynq Ultrascale+* [1] производства компании *AMD-Xilinx*), которая труднодоступна в России как из-за высокой стоимости и схемотехнической сложности применения [2, 3], так и из-за известных санкционных ограничений. В связи с этим являются актуальными теоретическая оценка и экспериментальная проверка возможности видеовывода сверхвысокой четкости (ВСВЧ) – *2K* и выше – с помощью микросхем более раннего семейства *Zynq-7000* [4].

Анализ проблемы

В самом простом случае аппаратная часть видеоподсистемы может состоять из самой микросхемы *Zynq-7000* и коннектора *HDMI*. В этом случае необходимые для работы подсистемы узлы обработки данных реализуются непосредственно на кристалле *Zynq* путем использования логических ресурсов *FPGA*. Один из вариантов реализации приведен на рис. 1, его основой является модуль *VDMA* [5].

С формальной точки зрения настройками *IP*-ядер *Xilinx* может быть задано высокое разрешение изображения (*8K* и даже выше), а главным ограничивающим фактором становится пропускная способность выходов *Zynq* (в частности, в наиболее доступных чипах этого семейства мультигигабитные трансиверы (*GTP, GTX*) отсутствуют [4]).

В соответствии с документацией [6], выводы банков *HD* (*High Density*) обеспечивают максимальную скорость передачи данных 1080 Мбит/с. В соответствии со спецификацией интерфейса *HDMI* [6, 7], используется дифференциальный интерфейс *TMDS* [8]. Для младших чипов семейства *Zynq-7000* максимальная теоретически достижимая скорость передачи данных по трем каналам *TMDS* достигает 3240 Мбит/с. Четвертый канал передает сигналы синхронизации.

Для оценки скорости передачи данных, требуемой для обеспечения заданного разрешения изображения и частоты кадров, необходимо обратиться к документации [9]. Количество активных (несущих информацию об изображении), вспомогательных (за время действия которых передается служебная

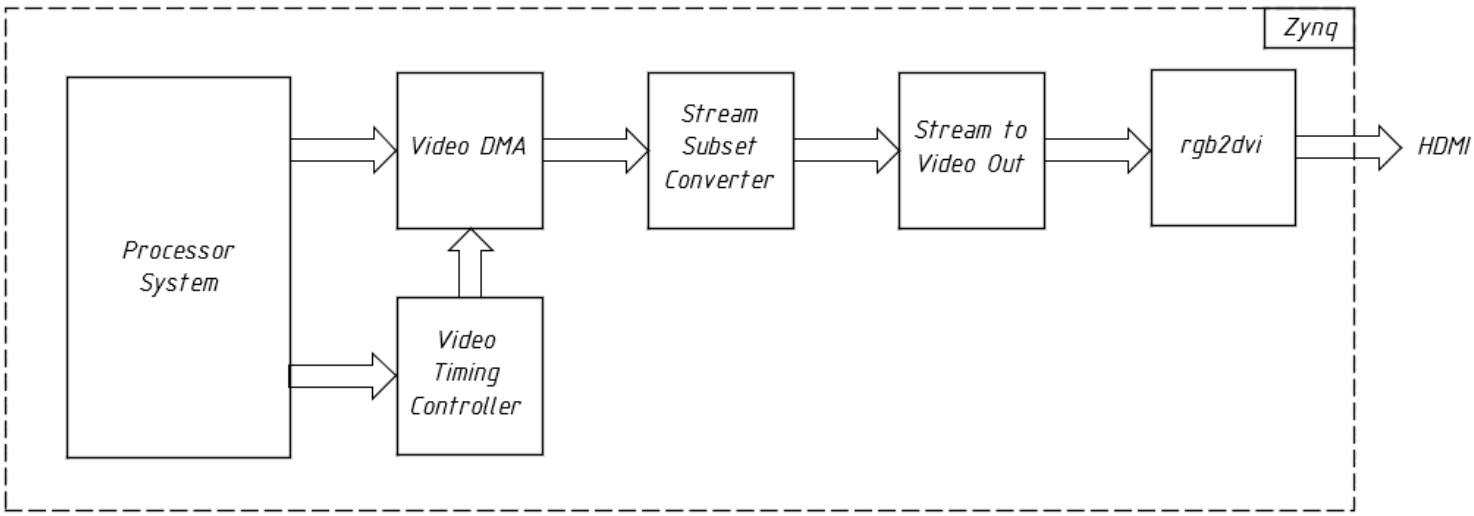


Рис. 1 – Функциональная схема видеоподсистемы *Zynq-7000*

информация, например, сигналы синхронизации) тактов и их общее количество нормируется для каждого режима работы дисплея (количество пикселей по горизонтали и строк по вертикали, а также частота кадров, иногда называемая также «частотой обновления»). Порядок определения параметров сигналов синхронизации также указан в [9]. Данный документ определяет три варианта временных параметров синхросигналов (полный и два сокращенных). Пример приведен в табл. 1. У сокращенных вариантов длительность синхросигналов меньше, поэтому общее количество пикселей также меньше. Тем самым снижаются требования к скорости передачи данных. Современные мониторы обычно справляются с отображением изображения по любому из стандартов, если его параметры не превышают возможности самого монитора.

Табл. 1 – Параметры синхронизации для видеорежима 3440x1400, 60 Гц, для различных стандартов

Параметр	Стандарт		
	CVT	CVT-RB	CVT-RB V2
Общее количество пикселей по горизонтали	4688	3600	3520
Количество активных пикселей по горизонтали	3440	3440	3440
Начало горизонтального синхроимпульса, номер пикселя	3696	3488	3448
Окончание горизонтального синхроимпульса, номер пикселя	4064	3520	3480
Полярностьгоризонтального синхроимпульса	отрицательная	положительная	положительная
Общее количество строк по вертикали	1493	1481	1481
Количество активных строк по вертикали	1440	1440	1440
Начало вертикального синхроимпульса, номер строки	1443	1443	1467
Окончание вертикального синхроимпульса, номер строки	1453	1453	1475
Полярность вертикального синхроимпульса	положительная	отрицательная	отрицательная
Частота синхронизации, МГц	419,5	319,75	312,787
Максимальная скорость передачи данных, Мбит/с	10068	7674	7507

Для быстрой оценки реализуемости того или иного видеорежима при заданной скорости передачи данных можно воспользоваться онлайн-инструментом [10]. Из таблицы следует, что можно достичь высоких разрешений (даже 8К) на довольно медленном канале (до 4 Гбит/с), но частота кадров составит всего несколько Герц, что в известной мере ограничивает практическое применение такого решения.

Предлагаемое решение

Для экспериментальной проверки приведенных положений авторы использовали плату собственной разработки (рис. 2) на базе микросхемы *XC7Z010-1CLG400C* [4], работающей совместно с динамическим ОЗУ *DDR3*. Плата предназначена для управления промышленным оборудованием (станком с ЧПУ) и имеет 14 слоев. Разъем *HDMI* подключен непосредственно к выводам микросхемы. В качестве операционной системы использовалась специализированная сборка *Petalinux 2020.2*, включающая в себя файлы видеодрайверов, модифицированные для использования ВСВЧ.

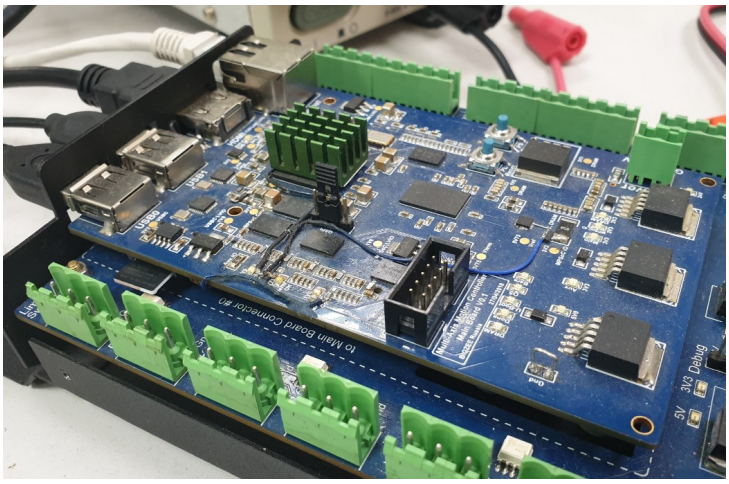


Рис. 2 – Плата на основе микросхемы *XC7Z010-1CLG400C,* применявшаяся в ходе экспериментов

В распоряжении авторов имелись два монитора, которые можно отнести к устройствам ВСВЧ – *Samsung LC27JG50QQIXCI* (разрешение до 2560 x 1440 при частоте кадров 60 Гц) [11] и *Xiaomi XMMNTWQ34* (разрешение до 3440 x 1440 при частоте кадров 60 Гц) [12]. Все эксперименты проводились с этими мониторами.

Экспериментальная установка показана на рис. 3. Использован высококачественный соединительный кабель стандарта *HDMI 2.1*, допускающий передачу изображения с разрешением *8K* при частоте кадров 60 Гц.

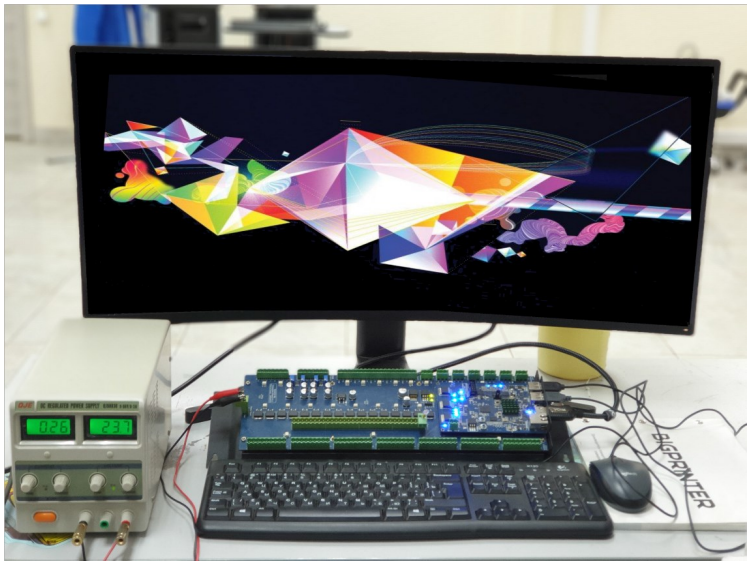


Рис. 3 – Внешний вид экспериментальной установки (выводится изображение с разрешением 3440 x 1440 пикселей)

В ходе экспериментов были последовательно заданы несколько видеорежимов (табл. 2). Фактическая частота кадров указана в таблице согласно техническим данным, отображаемым самим монитором.

Табл. 2 – Испытанные в ходе экспериментов видеорежимы

Горизонтальное и вертикальное разрешение, пикселей	Частота кадров, Гц
1920 x 1080	60
1920 x 1200	50
1920 x 1440	41
2048 x 1080	52
2160 x 1200	44
2560 x 1440	31
3440 x 1440	23
3840 x 2160	14

Заключение

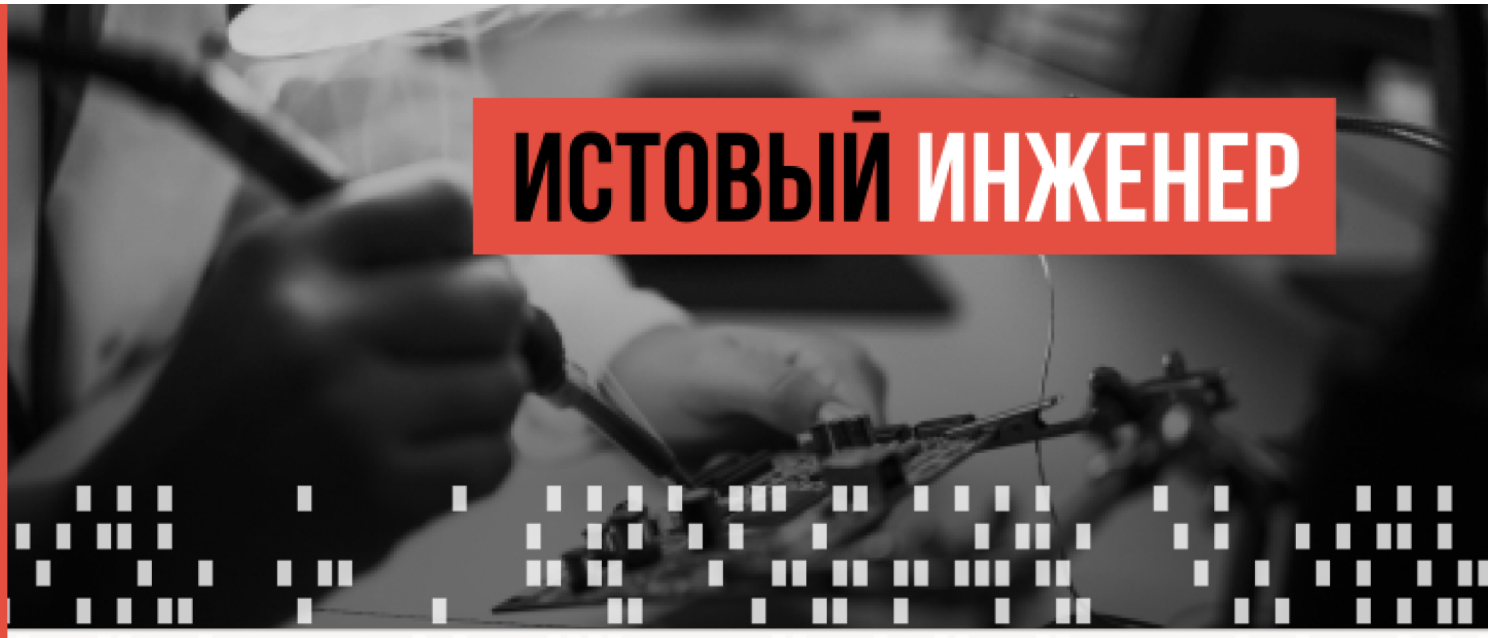
Проведенные расчеты и эксперименты подтвердили возможность реализации ВСВЧ на микросхемах *Zynq-7000* даже без использования гигабитных трансиверов. В то же время, такое решение имеет ограниченное применение, так как генерация изображения высокого разрешения требует значительных вычислительных ресурсов, а максимальная пропускная способность каналов *TMDS* на младших микросхемах *Zynq*-7000 не

позволяет получить частоту кадров выше примерно 20..25 Гц для разрешения 3440 x 1440.

Одним из вариантов применения этого решения является отображение графического пользовательского интерфейса промышленного оборудования. Такой интерфейс может иметь большое количество визуальных элементов, требующее ВСВЧ, но высокая частота обновления изображения при этом может и не требоваться, так как большая часть информации на экране статична (надписи, элементы дизайна), либо обновляется относительно редко (числовые и текстовые данные), так что частота кадров около 20 Гц вполне допустима.

Список источников

1. Zynq UltraScale+ MPSoC. Product Tables and Product Selection Guide. XMP104 (v2.6). AMD-Xilinx, 2022. 9 p.
2. Тарасов И. Е. Системы на модуле Kria компании Xilinx // Компоненты и технологии. 2021. № 6. С. 80–84.
3. Попов М. А., Машковская М. С., Романов А. Ю. Применение Xilinx Kria как базового модуля встраиваемой системы управления для промышленного оборудования // Информационные технологии. 2023 № 2. Вып. 29. С. 98–103.
4. Zynq-7000 SoC. Product Selection Guide. XMP097 (v1.3.2). AMD-Xilinx, 2019. 11 p.
5. AXI Video Direct Memory Access. LogiCORE IP Product Guide. PG020 (v6.3). AMD-Xilinx, 2022. 87 p.
6. Implementing a TMDS Video Interface in the Spartan-6 FPGA. XAPP495 (v1.0). Xilinx, 2010. 16 p.
7. High-Definition Multimedia Interface Specification Version 1.3a. Hitachi, Matsushita Electric Industrial Co., Philips Consumer Electronics, Silicon Image, Sony Corporation, Thomson, Toshiba Corporation, 2006. 276 p.
8. Spartan-6 FPGA SelectIO Resources. User Guide. UG381 (v1.7). Xilinx, 2015. 98 p.
9. VESA Coordinated Video Timings (CVT) Standard (v1.2). Video Electronics Standards Association, 2013. 26 p.
10. Verbeure T. Video Timings Calculator [Электронный ресурс]. – Электронные данные. – Режим доступа: https://tomverbeure.github.io/video_timings_calculator, свободный.
11. Samsung, Inc. 27″ Gaming monitor CJG5 [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.samsung.com/ru/monitors/gaming/wqhd-curved-monitor-with-144hz-refresh-rate-27-inch-lc27jg50qqixci, свободный.
12. Xiaomi, Inc. Mi Curved Gaming Monitor 34". [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.mi.com/ru/monitor34, свободный.



ИСТОВЫЙ ИНЖЕНЕР

Проект **«Истовый инженер»** рассказывает о технологиях и инженерной культуре, формируя новое понимание профессии российского инженера.

Для одних он станет качественным источником знаний про современный мир, который уже невозможно представить без сложных устройств и высоких технологий. Для других — мостиком в новую специальность.

Авторы материалов — ученые, предприниматели, инженеры. Эксперты в своей области, которые делятся знаниями и практическим опытом.

Пишем про **разработку и производство микро- и радиоэлектроники**, а также про то, как прийти в инженерную профессию и развиваться в ней.

Технические **лонгриды, видеолекции и подкасты** — выбирайте удобный именно вам формат получения знаний.

Начинающий специалист, профессионал или просто пытливый читатель — каждый найдет **интересную и доступную информацию** для своего уровня.

ПРИСОЕДИНЯЙТЕСЬ

engineer.yadro.com

Telegram-канал

YouTube

Реализация интерполятора на платформе SDR Pluto+

Афанасьев Никита, *neekeetos@gmail.com*

Телеграм @neekeetos

Обсуждение и комментарии :: ссылка

Описание задачи и аппаратной платформы

Платформа SDR Pluto+ (рис 1) это одноплатный компьютер (Single Board Computer ,SBC) построенный на основе SOC Zynq 7010 (System On Chip, SOC) от Xilinx/ AMD, содержащей 2х ядерный процессор ARM cortex-A9 и ПЛИС , работающий под управлением ОС Linux. Кроме базовой периферии на данной плате также присутствует цифровой трансивер AD9361, обеспечивающий прием и передачу произвольных сигналов двух независимых каналов в диапазоне 75МГц-6ГГц и полосой до 56МГц в режиме полного дуплекса. Трансивер использует параллельный интерфейс для передачи данных приема и передачи и подключается непосредственно к ПЛИС части, где работа с данными может быть реализована в зависимости от потребностей пользователя.

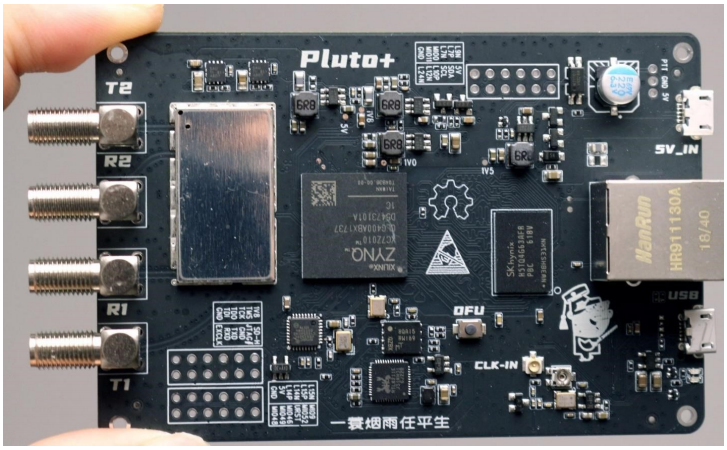


Рис 1. SDR платформа Pluto+ (фото из интернет)

Постановка задачи

В данном случае потребовалось реализовать передачу сигнала с ограничением его полосы в диапазоне 0,5-15МГц. Для реализации была выбрана схема представленная на (рис 2). В трансивере присутствует два канала передачи для каждого из которых необходимо сформировать поток данных (12бит x 2, комплексные отсчеты) с частотой оцифровки 65МГц. С целью оптимизации аппаратной части, частота оцифровки трансивера зафиксирована на 65МГц, а задача адаптации частоты оцифровки входного потока данных решается блоком интерполятора, который позволяет увеличить частоту оцифровки входного потока в диапазоне 4-64 раз, одновременно формируя фильтром интерполятора требуемую полосу сигнала. Блок интерполятора обрабатывает одновременно два канала с комплексными отсчетами, и фактически содержит 4 независимых интерполятора – по одному на каждую компоненту комплексных входных сигналов (I0 Q0 I1 Q1). Данные для передачи одновременно для двух каналов с комплексными отсчетами в формате 16 бит x2 x2 (16битные квадратуры для двух каналов, всего 4 слова по 16 бит), поступают AXI потоком из памяти SoC при помощи блока DMA. Блок интерполятора формирует на каждый входной отсчет сигнала M 16 битных выходных отсчетов для каждого из каналов и помещает их в выходной поток(где M - это коэффициент интерполяции) . Управление блоком интерполятора осуществляется программно через OS Linux, имеется возможность задания параметров ядра, например коэффициента интерполяции и произвольного набора коэффициентов фильтра.

Исходя из требований к ширине переходной полосы фильтра, расчетная длина фильтра оказалась равной ~20 коэффициентов на (выходной) отсчет для полифазного фильтра (это соответствует общей длине фильтра 80 коэффициентов для M=4 и 1280 коэффициентов для M=64), что при реализации «в лоб» потребует 20*4 = 80 умножителей/DSP блоков для обработки всех каналов на частоте равной частоте оцифровки. Такое количество умножителей составляет 100% от доступного количества умножителей и реализовав такой интерполятор остальные элементы схемы уже реализовать не удастся, в связи с этим потребовался более экономный вариант интерполятора.

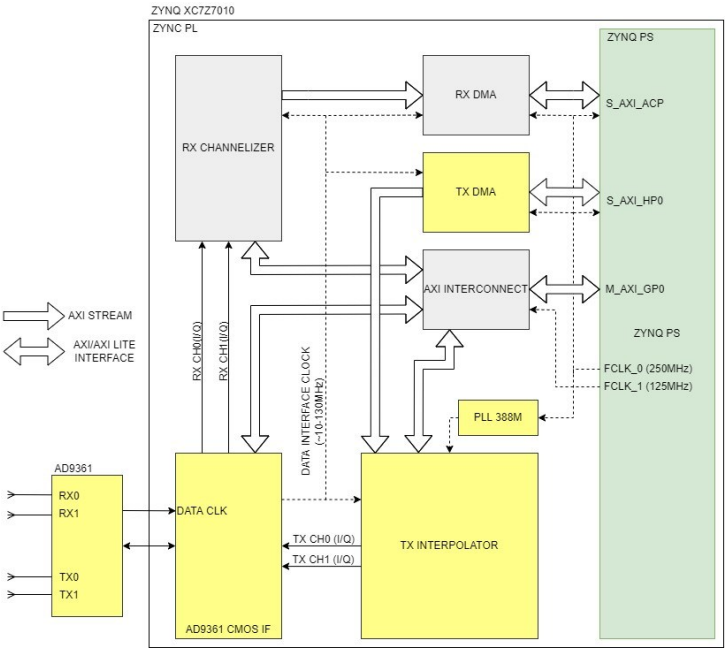


Рис 2 Общая структурная схема для задачи.

Рассуждения о ресурсах, экономный интерполятор.

Как было ранее выяснено, для нормальной работы интерполятора в реальном времени требуется производить 20 умножений на каждый отсчет выходного сигнала, самих сигналов 4. Можно оценить общий объем требуемых вычислений (считая только умножения) –

S = Fs * Nch * T,

где Fs частота оцифровки для выходных отсчетов, Nch – количество каналов, T - требуемое количество умножений на отсчет выходного сигнала (величина равна длине фазы полифазного фильтра, а количество фаз равно коэффициенту интерполяции и общая длина фильтра - T * M). Как видно объем вычислений не зависит от коэффициента интерполяции, удобно. Попробуем вычислить для нашего случая -

S = 65MHz * 4 * 20 = 5200M умножений в секунду.

Наиболее оптимальный вариант с точки зрения экономии ресурсов – это заставить работать DSP блоки на предельной для них частоте, это согласно описанию на ПЛИС часть используемой SOC Zynq 7010, для 1 спидгрейда составляет 464МГц, однако для данной задачи данные для расчета и коэффициенты фильтра придется загружать из блоков BRAM, которые работают максимум до 388МГц. Поэтому выберем данную частоту как рабочую, и оценим требуемое количество DSP блоков для работы фильтра, поделив объем требуемых вычислений на частоту работы одного блока –

Ndsp = 5200M / 388M = 13.4,

округляя это число до большей степени двойки получим 16 умножителей/DSP блоков, что дает нам в случае успешной реализации экономию в 80/16 = 5 раз по умножителям и займет уже не 100% а всего лишь 20% от доступных на кристалле DSP блоков ПЛИС.

Немного про интерполяцию сигнала (в целое число раз)

Интерполяция сигнала состоит из двух независимых процессов (которые могут быть объединены) – увеличение частоты дискретизации и фильтрация. Увеличение частоты достигается дополнением входного сигнала (M-1) нулями для повышения частоты оцифровки в M раз. При этом спектр получившегося сигнала будет содержать M копий спектра входного сигнала. Для получения интерполированного сигнала все копии спектра, кроме исходной (на 0 частоте) отфильтровываются. Для фильтрации используется КИХ

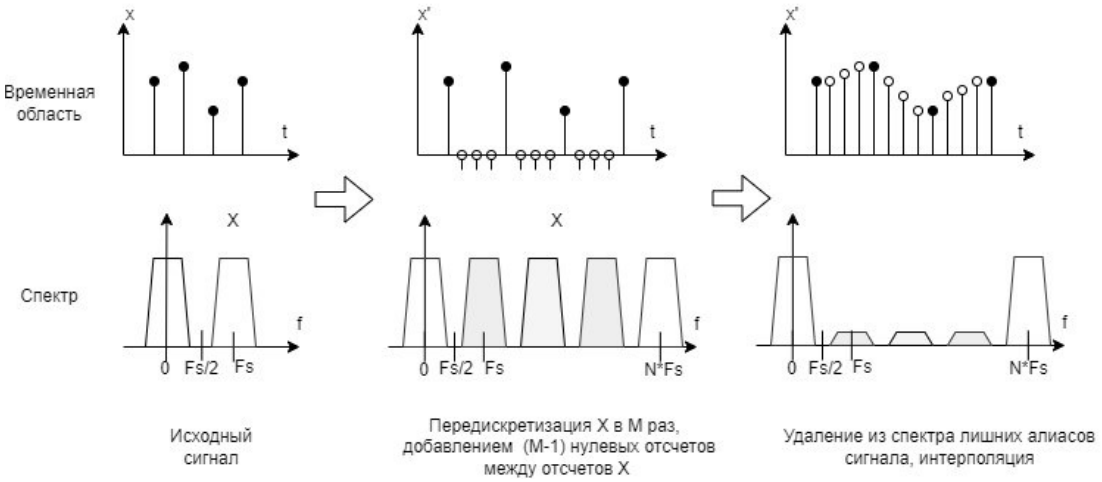


Рис 3 Сигналы интерполятора на разных этапах, второй этап обычно используется для пояснения работы полифазного фильтра интерполятора, и причин его использования

фильтр. Схематично данный процесс представлен на (рис 3)

Полифазный фильтр строится на основе КИХ фильтра со следующей формулой:

$$y(n) = \sum_{i=0}^{N} h(i) * x(n-i),$$

где N общая длина фильтра, x(n) отсчеты входных данных, y(n) − выходной отсчет фильтра n .

Разобьем данную сумму на M групп, для N кратного M:

$$y(n) = \sum_{r=0}^{M-1} \sum_{i=0}^{\frac{N}{M}-1} h(M*i+r) * x\big(n-(M*i+r)\big) \qquad (1)$$

На этапе передискретизации в M раз в исходный сигнал добавляются нули, получившийся сигнал представляет из себя следующую последовательность:

$$X(M*n^*+r) = \begin{cases} x(n^*) & \text{для } r = 0 \\ 0 & \text{для } r = 1..M-1 \end{cases} \qquad (2)$$

Где n* является индексом входной последовательности. Если подставить (2) в (1) , получим

$$y(n) = \sum_{r=0}^{M-1} \sum_{i=0}^{\frac{N}{M}-1} h(M*i+r) * X\big(n-(M*i+r)\big) \qquad (3)$$

Согласно (2) X(n) имеет ненулевые отсчеты только если n кратно M, следовательно индекс X

$$n-(M*i+r) \qquad (4)$$

Должен быть кратен M и величина (n-r) = kM, для данного индекса отсчета n существует лишь одно значение r из диапазона 0…M-1, для которого (n-r) кратно M. Можно упростить выражение (1), записав его для индексов исходного сигнала и учитывая, что сумма по r имеет лишь одно значение r с ненулевой суммой:

$$y\ \ (Mn^*+r) = \sum_{i=0}^{\frac{M}{N}-1} h(M*i+r) * x(n^*-i) \qquad (5)$$

Выражение (5) является формулой для вычисления значения отдельной фазы с номером r для полифазного фильтра интерполятора.

В данном S(n) это последовательность входных отсчетов x(n) дополненная нулями, h/Y(0)…h/Y(4) коэффициенты фильтра, расположенные напротив отсчетов , которые требуются для вычисления соответствующего выходного значения Y(0)…Y(4). Цветом выделены коэффициенты фильтра, соответствующие ненулевым элементам последовательности S(n).

Ключевые элементы аппаратной реализации интерполятора на ПЛИС

Как можно узнать из литературы, КИХ фильтр реализуется как сумма произведений сигнала на коэффициенты фильтра. Для вычисления одного выходного отсчета необходимо в общем случае сложить N произведений, где N − общее количество коэффициентов фильтра. В случае с полифазным фильтром длина суммы равняется длине одной фазы (для N кратного M), которая в свою очередь равна для интерполятора N/M где N общая длина фильтра, M - коэффициент интерполяции. Для аппаратной реализации удобно сразу создать блок, реализующий операцию умножения с накоплением (Multiply-Accumulate, MAC), и позволяющий обнулять сумму в случае необходимости. И на основе него построить элемент фильтра интерполятора согласно (рис 4).

Элемент фильтра (Filter unit) позволяет последовательно рассчитать часть фаз фильтра интерполятора для двух независимых входных каналов. Входной и выходной потоки данных используют интерфейс AXI Stream (поток AXI), при этом логика работы выходного потока не позволяет остановить работу интерполятора, в случае занятости потребителя данных (не реализован механизм back pressure). Блок реализован на двух MAC блоках и двух блоках BRAM (память входных данных XRAM и коэффициентов фильтра HRAM), также содержит логику генерации адресов для памяти(Address Generation Unit, AGU), контроля потока и выходной буфер для преобразования формата сигналов управления из стробов в AXI поток (с оговоркой выше). В блоке логики

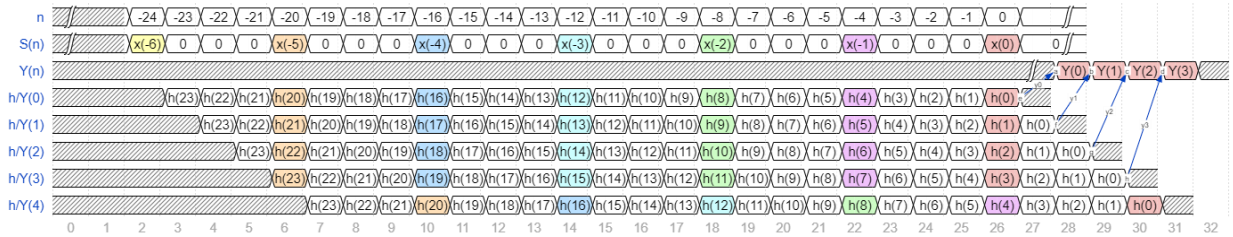


Рис 3.1 Пример расчета полифазного фильтра длиной N = 24, коэффициента интерполяции M=4

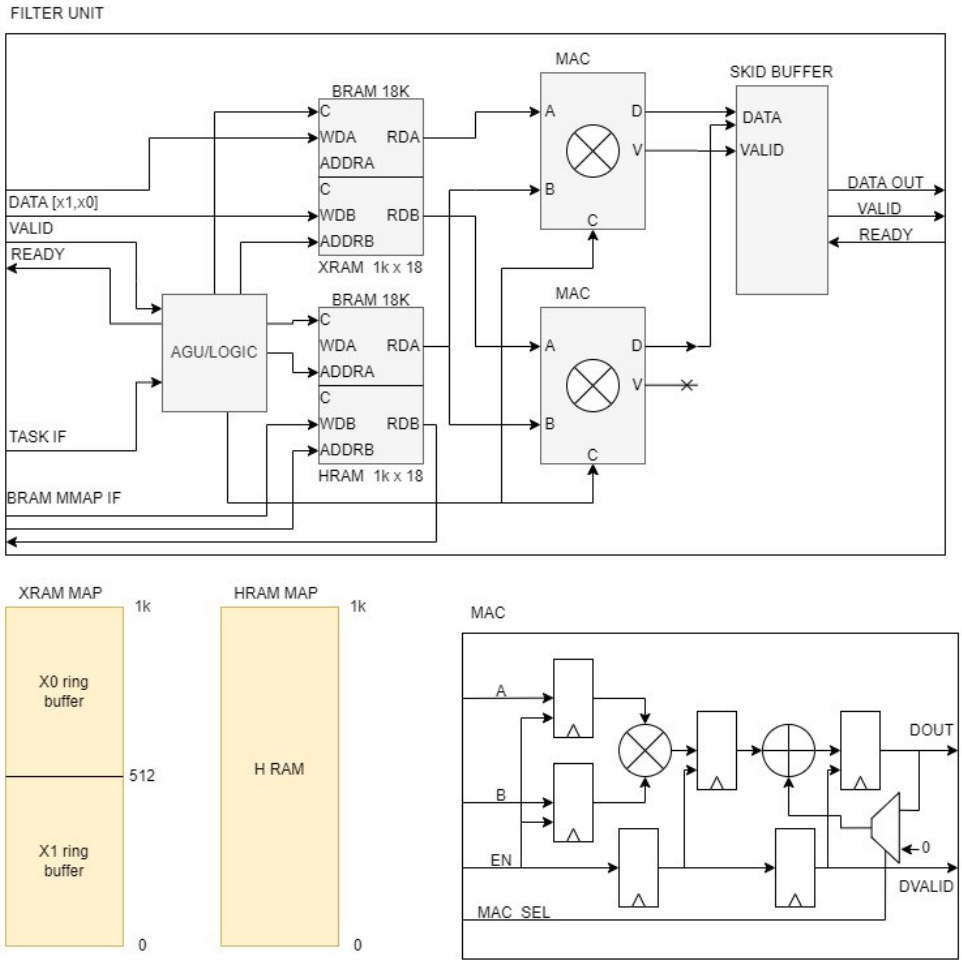


Рис 4. Элемент фильтра интерполятора с картой памяти коэффициентов и входных данных и ядром MAC.

также реализован интерфейс для настройки параметров фильтра (длина фазы, количество фаз). Данный блок конфигурируется двумя параметрами Mf (максимальное количество фаз) и Nf(максимальная длина фазы) на этапе разработки, при выборе оптимальной емкости фильтра, при этом Mf*Nf = 512 и должны быть степенями двойки. При этом в процессе работы можно задавать любые значение меньшие или равные этим величинам. Для максимального количества фаз Mf = 16 , максимальная длина фазы оказывается равной 32 (тк Mf*Nf = 512, то Nf = 512/16 = 32) , что выше чем требуется в задаче (20) .

Общая схема реализации интерполятора

В результате нескольких итераций появилась схема,

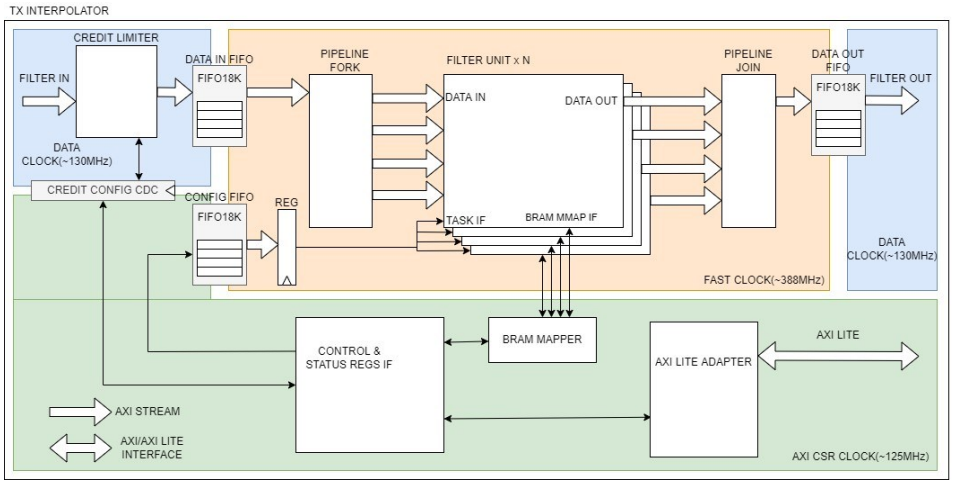


Рис 5. Схема реализации ядра интерполятора TX Interpolator.

реализующая блок интерполятора и использующая элемент фильтра как основу для реализации экономного интерполятора, она представлена на (рис 5).

Данный блок построен исходя из того, что фильтр интерполятор позволяет проводить расчет отдельных фаз параллельно. Реализация предусматривает создание нужного количества элементов фильтра, которые совместно реализуют расчет всех выходных данных. Общее количество фаз фильтра делится между отдельными элементами фильтра, каждый из которых считает свой независимый набор фаз (и общее максимальное количество фаз фильтра/коэффициента интерполяции становится M = Mf * N, где Mf максимальное количество фаз элемента фильтра, N – количество элементов фильтра , M в существующей реализации для Mf = 16 и N=4 составляет 64 , определяя максимальный коэффициент интерполяции, минимальный же определяется величиной N т.к. каждый элемент фильтра должен содержать минимум одну фазу для расчета). На схеме цветом отмечены разные зоны тактирования, в частности тактирование самого фильтра (FAST CLOCK 388MHz), интерфейса данных трансивера (DATA CLOCK 10-130МГц), интерфейса управления (AXI CSR CLOCK 125MHz). Переход между данными тактами осуществляется через аппаратные FIFO18к, это решение в том числе не позволяет эффективно использовать контроль потока данных из-за возникающей задержки и потребовало создание модуля кредитного буфера, который ограничивает поток входных данных в случае аварии на выходе.

Входные данные для обработки поступают со входа Filter IN и поступают на вход кредитного буфера, который занимается оценкой количества данных, находящихся в обработке внутри фильтра, для этого данных модуль также подключен к сигналам управления выходного потока данных. В случае, если количество данных в фильтре превышает порог (который задается программно), кредитный буфер перестает помещать данные на вход фильтра до тех пор, пока выходные данные фильтра не будут выгружены в достаточном количестве. После кредитного буфера данные поступают в фифо и на стороне фильтра оказываются в блоке PIPELINE FORK , который дублирует входной поток данных в несколько потоков – по одному для каждого элемента фильтра. Далее каждый блок осуществляет расчет каждый своего набора фаз фильтра интерполятора и помещает их в выходной поток. Модуль PIPELINE JOIN осуществляет объединение выходных потоков данных с отдельных элементов фильтра в общий поток и помещает его в выходное фифо, которое переносит данные обратно в зону тактирования интерфейса данных, где они поступают на вход блока интерфейса трансивера и далее передаются в трансивер для передачи.

Заключение и выводы

В рамках проделанной работы удалось реализовать блок интерполятора, с существенной экономией ресурсов ПЛИС, методы, использованные для этого, включают реализацию полифазного интерполятора и конвейера обработки данных, работающего на частоте выше частоты оцифровки входного сигнала. Изложенный опыт может быть полезен при разработке устройств с аналогичным функционалом.

Литература

1. Frederic J Harris Multirate Signal Processing for Communication Systems 2nd ed. 2021.
2. L. R. Rabiner , B. Gold Theory and application of digital signal processing 1975.

PYNQ для систем-на-кристалле на примере реализации множества Мандельброта

В.В.Гуров, *кандидат технических наук преподаватель спецдисциплин Филиал РКТ МАИ в г.Химки* gurovvv@mai.ru , va1ery@ya.ru

Обсуждение и комментарии :: ссылка

Введение

Отладочные платы, содержащие программируемые логические интегральные схемы — ПЛИС, или FPGA (Field-Programmable Gate Array), на сегодняшний день известны, пожалуй, не меньше, чем платы Arduino лет десять тому назад. Семейство последних, кстати, в 2018 г. пополнилось платой MKR Vidor 4000 с FPGA-чипом Cyclone 10 компании Altera (в составе Intel).

Растущая популярность ПЛИС обусловила появление платформ и инструментов, позволяющих конфигурировать их без необходимости использовать для этого проприетарное ПО, вроде Vivado или Quartus. К таким платформам относится проект с открытым исходным кодом PYNQ [1] компании AMD, ориентированный на упрощение использования адаптивных вычислительных платформ на основе ПЛИС.

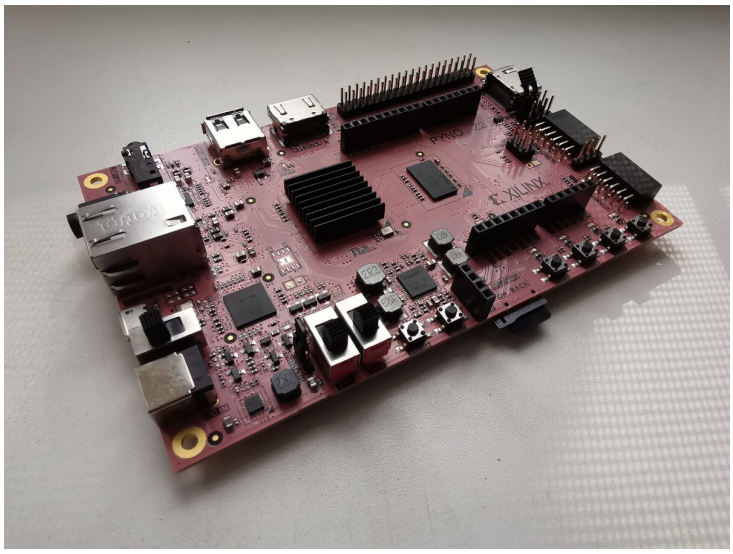
О проекте PYNQ

Опираясь на возможности языка Python, разработчики PYNQ могут использовать преимущества как программируемой логики (Programmable Logic, PL), так и встроенных микропроцессоров (Processing System, PS) для создания мощных и интересных электронных систем. Так, преимущество PL по сравнению с процессорными системами заключается в том, что логика приложения реализуется непосредственно аппаратными схемами PL, а не выполняется поверх ОС, драйверов и пр. В отличие от процессоров, ПЛИС самой своей природой реализует параллелизм, и различным операциям обработки данных не приходится конкурировать за аппаратные ресурсы.

На сайте проекта, в частности, сказано , что *«PYNQ поддерживает FPGA-платы с чипами Zynq, Zynq UltraScale+, Zynq RFSoC, платы ускорителей Alveo и AWS-F1 для создания высокопроизводительных приложений с параллельным аппаратным выполнением, обработкой видео с высокой частотой кадров, применением алгоритмов с аппаратным ускорением, обработкой сигналов в реальном времени, вводом-выводом с высокой пропускной способностью»*.

Фактически, PYNQ представляет собой кастомизированный образ ОС Linux, записываемый на карту памяти microSD, с которой осуществляется загрузка платы с поддержкой PYNQ. После загрузки такую плату можно легко запрограммировать в среде Jupyter Notebook, используя аппаратные библиотеки, или оверлеи (overlays), ускоряющие работу ПО.

Недорогая плата начального уровня PYNQ-Z2 оснащена чипом ZYNQ XC7Z020 компании XILINX (в составе AMD), представляющим собой «систему-на-кристалле», или system-on-chip (SoC), на базе двухъядерного процессора ARM Cortex-A9 (вышеупомянутая Processing System, PS), интегрированного со структурой FPGA (т.е. Programmable Logic, PL).

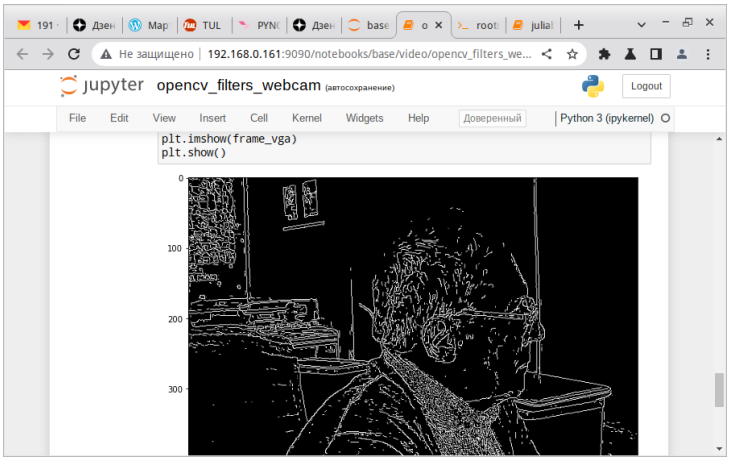


Плата PYNQ-Z2 с разъёмами Pmod и Arduino

Оверлеи, или аппаратные библиотеки представляют собой настраиваемые конфигурации FPGA, проще говоря, битстримы (bitstreams), расширяющие пользовательское приложение возможностями параллельной обработки, реализуемыми непосредственно «в железе». Оверлеи можно

использовать для ускорения работы ПО или в качестве заказной аппаратной платформы конкретного приложения.

Например, обработка изображений является типичным приложением, где технология FPGA способна обеспечить значительное ускорение. Программист может рассматривать оверлей как аналог библиотеки ПО, содержащий функции обработки изображений (например, определение краев, пороговых значений и т. д.), «зашитые» в FPGA.



Обработка изображений на ПЛИС

Оверлеи динамически загружаются в ПЛИС по мере необходимости, как и программная библиотека. Так, в данном примере отдельные функции обработки изображений могут быть реализованы даже в разных оверлеях и вызваться из них посредством API Python.

Фреймворк PYNQ, таким образом, предоставляет API, позволяющий управлять оверлеями в PL из среды Python, запущенной в PS. Конфигурирование же «фабрики» ПЛИС — это специализированная задача, требующая знаний и опыта в области аппаратного проектирования. Поэтому оверлеи PYNQ создаются инженерами-проектировщиками оборудования и затем «обертываются» посредством Python API. Благодаря этому разработчики ПО могут использовать интерфейс Python для управления специализированными аппаратными оверлеями, не создавая их самостоятельно. Это аналогично программной библиотеке, написанной опытным программистом, которую затем используют другие [2].

По умолчанию записываемый на microSD-карту и загружаемый в SoC образ PYNQ включает в себя так называемый базовый оверлей — Base. Его задача — обеспечить доступность имеющихся на плате периферийных устройств «из коробки». Поскольку базовый оверлей уже включает в себя блоки интеллектуальной собственности (Intellectual Property, IP) для всех периферийных устройств платы, он может служить в качестве прототипа для создания новых настраиваемых оверлеев, о чем будет рассказано далее.

Главное достоинство PYNQ, таким образом, заключается в возможности формирования библиотек пользовательских IP-ядер для FPGA и последующего запуска этих ядер с помощью Jupyter Notebook на платах SoC. Это делает внедрение решений на базе ПЛИС значительно более простым и доступным. Вокруг PYNQ уже образовалась довольно обширная экосистема, включающая множество проектов, нацеленных на обучение студентов, научные исследования в области ИИ, промышленных приложений, в числе которых разработка криптографических устройств, систем управления двигателями и робототехнических платформ.

"Железный" оверлей

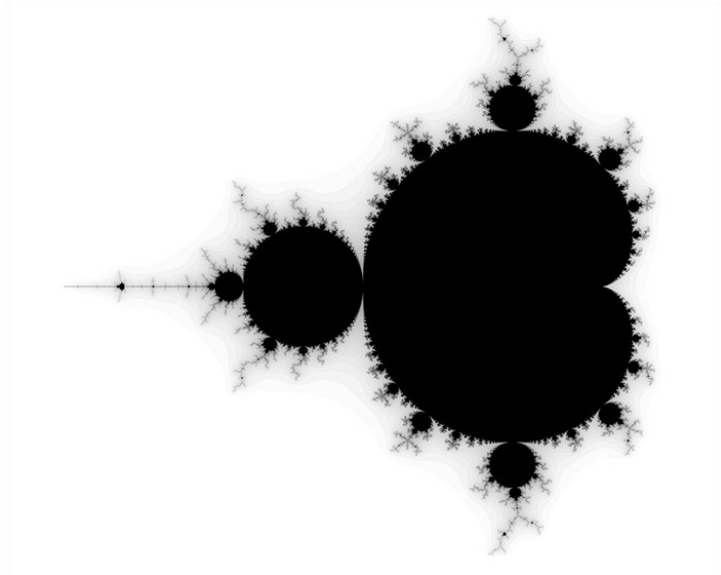
Среда Jupyter Notebook упрощает управление конфигурированием ПЛИС, но, как уже говорилось ранее, оверлеи PYNQ приходится создавать с помощью инструментов разработки оборудования. Поэтому далее речь пойдет о том, как практически с нуля написать свой оверлей и затем интегрировать его с "железом" ПЛИС.

Воспользуемся для этого чужим оригинальным проектом фрактального рендеринга, размещенным на Гитхабе в открытом доступе [3]. Как отмечает его автор, рендеринг фракталов представляет собой вычислительную задачу, требующую большой производительности для достижения удовлетворительных результатов. Для аппаратного ускорения алгоритма путем его реализации на ПЛИС автор использует возможности PYNQ, чтобы взаимодействовать с загруженным в ПЛИС оверлеем. Вот что пишет автор в своем отчете [4].

*"Множество Мандельброта определяется как набор комплексных чисел **c**, для которых функция (1) не расходится при итерации от **z=0**... Для достижения лучших визуальных результатов при вычислении фрактала отдельным пикселям сгенерированного изображения присваивается цвет в зависимости от того, сколько итераций требуется для превышения порогового значения,... что формирует цветовой градиент и создает интересные узоры. Пикселям, значения функции для которых не расходятся (не превышают порогового значения после определенного количества итераций), может быть присвоен специальный цвет, чтобы отличать их от остальных. Каждый пиксель может быть вычислен независимо, что позволяет аппаратно сгенерировать фрактал в виде потока независимых пикселей…".*

$$f_c(z) = z^2 + c \ (1)$$

Пример множества Мандельброта с цветным окружением в оттенках серого можно увидеть на рисунке.



Множество Мандельброта

Выбрав целью множество Мандельброта, в качестве отправной точки средствами высокоуровневого синтеза (High-Level Synthesys, HLS) автор скомпилировал блок интеллектуальной собственности (IP), способный генерировать фрактальные изображения на основе заданных входных данных. Сначала проект состоял из простого IP-ядра Calc_pixel, вычисляющего значение итерации для одного пикселя. Функция верхнего уровня в этом случае называется calc_pixel. В целом же процесс вычислений описывается автором следующим образом.

"Модель состоит из единственного класса Mandelbrot с главной функцией calculator, которая вычисляет фрактальное изображение по заданной конфигурации и возвращает результат. Алгоритм выполняет итерацию по всем пикселям на экране. Для каждого пикселя по его координатам действительной оси x_0 и мнимой оси y_0, итеративно вычисляются уравнения (2-5), до тех пор, пока не будет достигнуто максимальное количество итераций или пока значение x^2 + y^2 не начнет отклоняться от порогового значения, равного 4 (условие x ^ 2 + y ^ 2 > 4 истинно)".

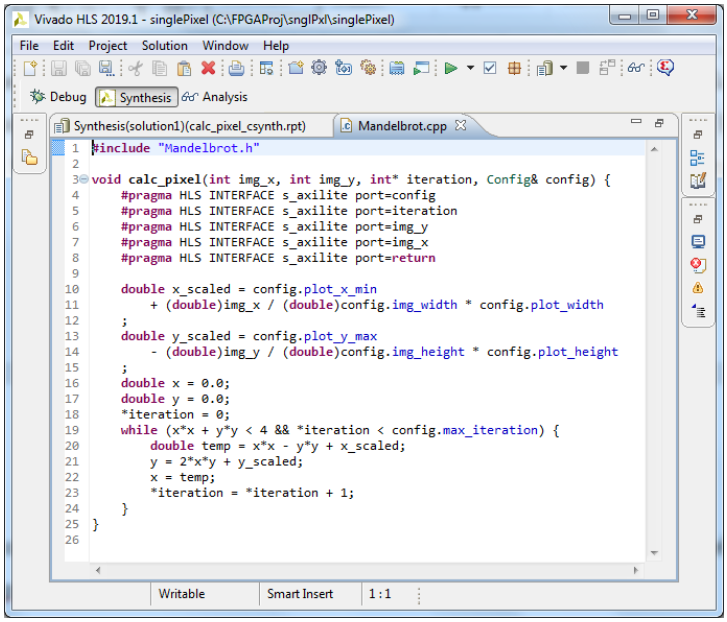
$$z^2 = (x + iy)^2 = x^2 + i2xy - y^2 \quad (2)$$

$$c = x_0 + iy_0 \quad (3)$$

$$x = Re(z^2 + c) = x^2 - y^2 + x_0 \quad (4)$$

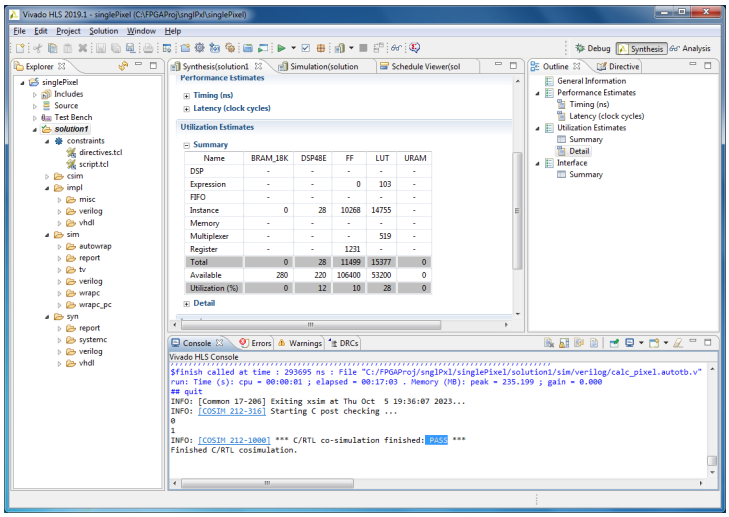
$$y = Im(z^2 + c) = 2xy + y_0 \quad (5)$$

Дополненный директивами Vivado соответствующий код на языке C++ приведен ниже.

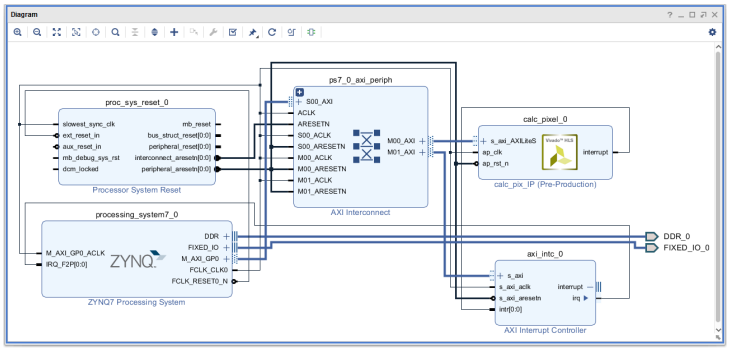


Функция верхнего уровня calc_pixel

После успешного синтеза и тестирования был выполнен экспорт IP-ядра. Затем непосредственно в среде Vivado IP Integrator был сформирован блочный дизайн всего оверлея, включающего в себя четыре библиотечных IP-блока и блок calc_pix_IP, синтезированный, как описано выше.



Результат синтеза и тестирования C/RTL

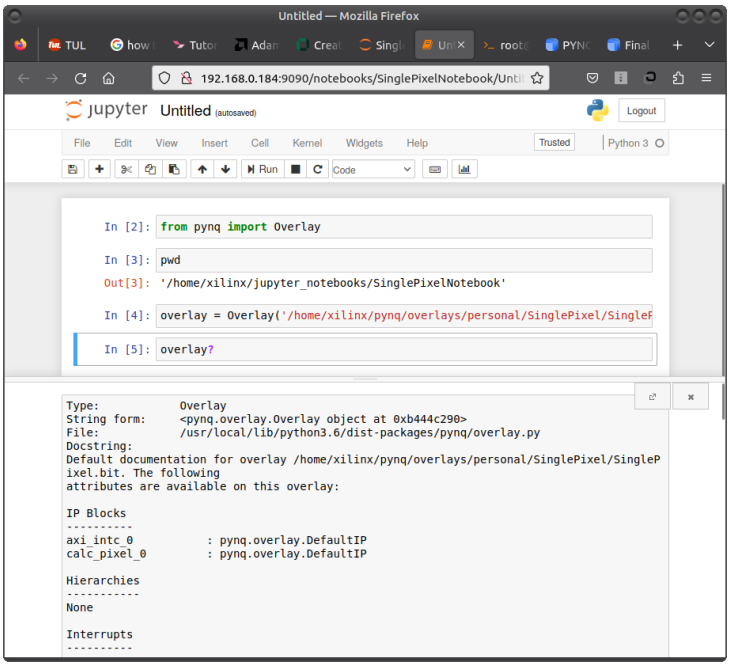


Блочный дизайн оверлея в Vivado IP Integrator

Сгенерированный Vivado файл битстрима с файлами сценария и описания оборудования затем были помещены в папку *personal* на microSD-карте PYNQ:

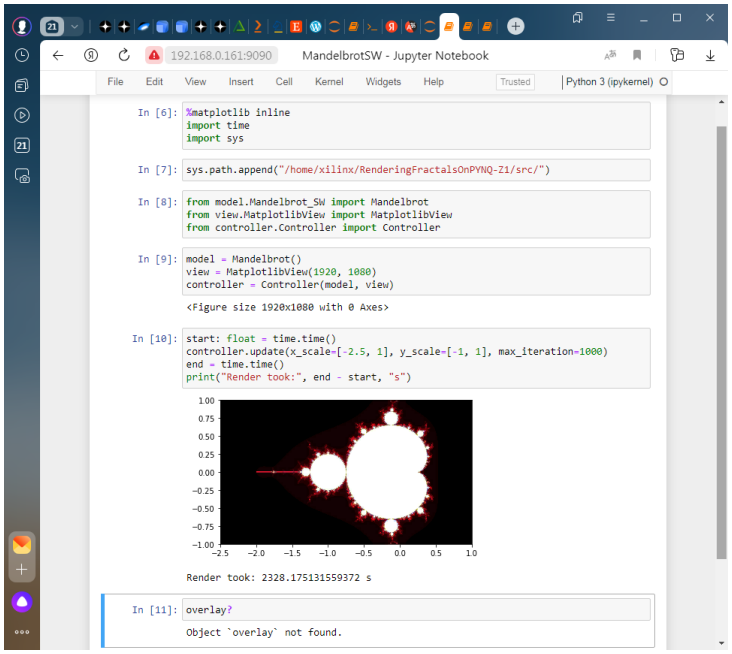
```
root@pynq:/home/xilinx/pynq/overlays/personal# ls -l SinglePixel/SinglePixel.*

-rw-rw-r-- 1 root xilinx 4045676 Oct 7 14:03 SinglePixel/SinglePixel.bit

-rw-r--r-- 1 root xilinx 276896 Oct 7 13:54 SinglePixel/SinglePixel.hwh

-rw-rw-r-- 1 root xilinx 2360 Oct 7 14:01 SinglePixel/SinglePixel.tcl

root@pynq:/home/xilinx/pynq/overlays/personal#
```

Теперь настало время выполнить т.н. Smoke Test, т.е. в данном случае проверить, «видит» ли Python наш первый оверлей.



Python «увидел» наш оверлей!

В списке IP-блоков мы находим необходимую нам аппаратную функцию calc_pixel_0. Теперь напишем другой Notebook, точнее позаимствуем упрощенную версию из отчета автора исходного проекта, не забыв также про файлы модели, отображения и контроллера (соответственно Mandelbrot_HW.py, MatplotlibView.py и Controller.py). После копирования файлов на microSD жмем на Cell/Run All и по прошествии четверти с лишним часа (!) получаем долгожданную картинку, правда, в инвертированном виде.



Наш первый "железный" фрактал

Несмотря на то, что завершение рендеринга заняло аж 994,13 секунды, а у автора оригинального проекта – 1025,3 с, полученный результат, указывает автор, с одной стороны, может служить доказательством успеха реализации кода с аппаратным ускорением, правда, пока явно недостаточным. С другой стороны, эту начальную стадию необходимо пройти, чтобы лучше понимать концепции IP, оверлеев и аспектов проверки правильности их проектирования, прежде чем переходить к более сложному и "быстрому" решению.

В ПЛИС все параллельно

Далее перескажем вкратце, что предпринял автор проекта для его успешного завершения. Чтобы ускорить обработку и избежать ненужного взаимодействия с кодом Python во время выполнения, IP-блок должен самостоятельно перебирать пиксели и выводить все отрисованное изображение целиком. Есть два варианта, как этого можно достичь: либо записать массив значений итерации в память, используя IP-блок прямого доступа к памяти (AXI Direct Memory Access, DMA), перед обработкой и отображением картинки с помощью кода Python, либо передать уже полностью обработанный кадр в блок прямого доступа к памяти AXI Video (VDMA).

Для PYNQ предусмотрены библиотеки для взаимодействия с VDMA и обработки видео, включая ввод-вывод HDMI. Это открывает возможность потоковой передачи данных на выход HDMI, который присутствует на устройстве, сохраняя при этом возможность считывать кадры и в Jupyter notebook. Базовый оверлей, как уже говорилось выше, доступен для скачивания вместе с названными библиотеками внутри репозитория PYNQ на странице Xilinx на GitHub. Далее будет использоваться подход, основанный на VDMA.

Основной файл Mandelbrot.cpp должен быть полностью переписан. Эти изменения, в свою очередь, должны быть отражены в файлах заголовка и тестбенча.

Библиотека hls_video.h включена в файл Mandelbrot.h, содержащий определения типов шаблонов, используемых для обработки видео. Требуемыми шаблонами являются hls::stream для определения выходных данных, hls::Mat для определения матрицы хранения выводимых данных и hls::Scalar для определения фрагмента данных в матрице.

Затем определяются новые типы, основанные на включенных типах шаблонов, с их описательными названиями, где AXI_STREAM представляет 32-разрядный потоковый интерфейс AXI, используемый для вывода изображения из синтезированного IP-блока в VDMA, RGB_IMAGE - изображение 1080p, которое может быть выведено через этот поток, а RGB_PIXEL - отдельный пиксель этого изображения, содержащий три 8-битных цветовых канала.

Функционально файл IP-блока Mandelbrot.cpp разделен на три отдельных метода, чтобы улучшить общую читаемость кода.

```
void mandelbrot(AXI_STREAM& OUTPUT_STREAM, Config& con-
fig)
```

Функция верхнего уровня, которая обрабатывает ввод и вывод.

Определяется новая переменная img, которая затем передается функции calculate для обработки перед выводом в OUTPUT_STREAM.

Тип вывода AXI_STREAM вместе с pragma DATA FLOW позволяет передавать пиксели по мере готовности из IP-блока один за другим в переменную img для повышения производительности.

```
void calculate(Config& config, RGB_IMAGE& img)
```

Содержит исходный код обработки, заключенный в два цикла for.

Результат записывается в переменную img, предоставленную в качестве аргумента.

```
RGB_PIXEL getPixel(int& iteration, int& max_iteration)
```

Возвращает RGB_PIXEL на основе предоставленного значения итерации.

Цвет меняется с черного на зеленый, затем на белый и так повторяется по кругу.

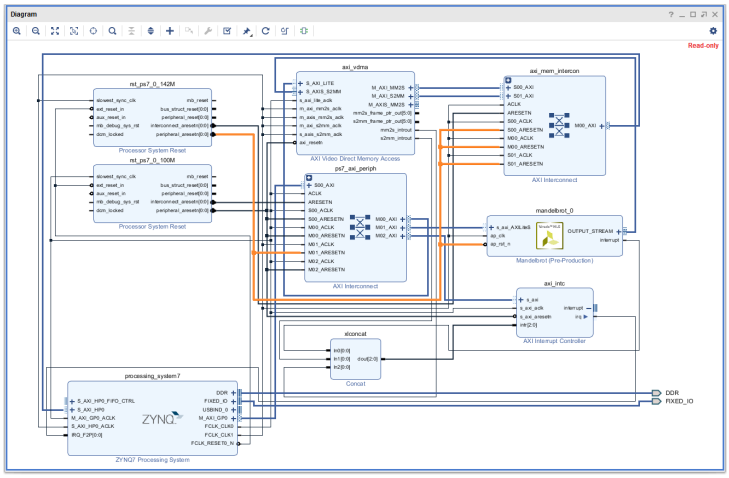
В особом случае, когда достигается значение max_iteration, выводится черный цвет.

Выполняет интерполяцию между 512 различными оттенками зеленого.

Все исходные коды приведены полностью в отчете [4] и на Гитхабе [3].

С появлением потоковых интерфейсов и добавлением блока памяти прямого доступа к видео дизайн оверлея усложняется. Потоки AXI4 работают на более высокой тактовой частоте, и поэтому в конструкцию необходимо добавить другой источник тактовой частоты наряду с собственным блоком сброса системы процессора. Это можно видеть в конструкции базового оверлея, а также в спецификации IP-блока AXI VDMA, которая определяет максимальную частоту для AXI4-Stream как 150 МГц.

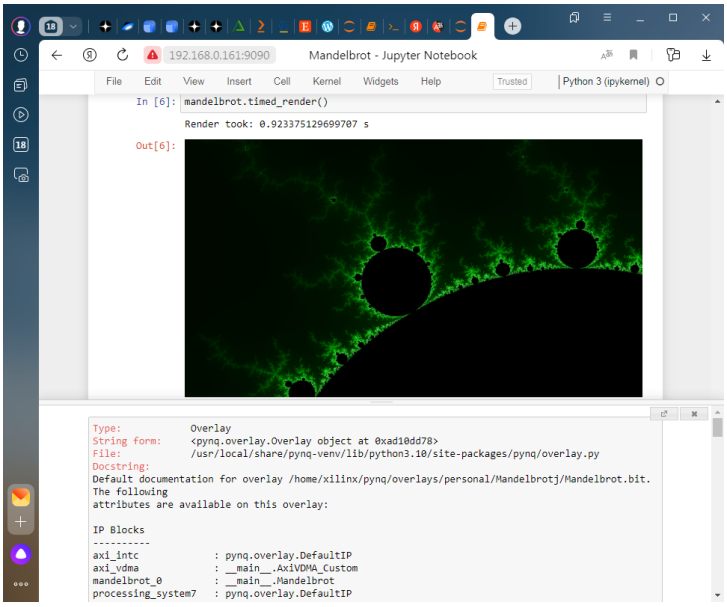
В базовом оверлее для платы PYNQ-Z1, на которую ориентирован оригинальный проект, внутри иерархии видео потоки AXI передаются через иерархии HDMI. Проанализировав иерархию hdmi_out и ее компоненты, предоставленные вместе с базовым оверлеем, автор выяснил, что все они имеют свойство clk_period, указанное равным 7 нс, что соответствует примерно 142,85 МГц. Но даже если IP-блок рассчитан на частоту 100 МГц, которая является стандартной предустановкой в Vivado HLS (тактовый цикл 10 нс), он все равно сможет работать на частоте 142 МГц без каких-либо проблем. На этом этапе проектирования целевой тактовый период для IP-блока Mandelbrot установлен на 7 нс, чтобы позволить ему и блоку VDMA работать на более высокой тактовой частоте 142 МГц.



IP-блок Mandelbrot может работать на тактовой частоте 142 МГц

Далее повторяются все уже известные нам шаги: выполняется генерация битстрима, файлы Vivado копируются на microSD, для поддержки нового оверлея дописываются классы модели, отображения и контроллера, а также Jupyter Notebook [3].

И вот наступает следующий долгожданный момент! Изображение фрактала целиком (а не попиксельно с передачей каждого пикселя в Python!) генерируется "железом" ПЛИС и только после этого выводится на экран.



Завершение рендеринга заняло меньше секунды!

Сравните: 994,13 и 0,92337 секунды! Тысячекратное ускорение обеспечено программируемой логикой FPGA. Включение директивы #pragma HLS PIPELINE во внутренний цикл функции calculate позволило конвейеризовать обработку и, таким образом, увеличить пропускную способность. Благодаря этому каждое вычисление, выполняемое на любой отдельной итерации внутреннего цикла, не зависело от результатов, полученных на нескольких предыдущих итерациях, и обработка текущей итерации продолжалась без необходимости ждать завершения выполнения предыдущих итераций, все еще находящихся в конвейере.

В заключение вот, что пишет автор в своем отчете [4].

"Вычисление значения итерации каждого пикселя является независимым. Это означает, что вычисление может выполняться для любого количества пикселей одновременно. Это может быть использовано для создания нескольких копий одного и того же процессингового оборудования, чтобы увеличить скорость, с которой IP-блок может выдавать изображения, не оказывая никакого влияния на вычисления. Рассматриваемый IP-блок обрабатывает полный ряд пикселей за раз. Цель состоит в том, чтобы иметь возможность обрабатывать несколько строк одновременно, в то же время позволяя разбивать отдельные строки на разделы, где каждый раздел в пределах любой строки может быть обработан собственной копией "железа". Строка пикселей достаточно длинная, и ее можно разделить на несколько секций, не нарушая согласованности конвейера, что позволяет увеличить параллелизм IP-блока без увеличения и без того высоких объемов BRAM, потребляемых массивами, что в противном случае ограничило бы количество копий процессингового оборудования, которые могли бы быть созданы, поскольку на них просто не хватит памяти".

Список источников

1. Python productivity for Zynq [Электронный ресурс] http://www.pynq.io/, дата обращения 14 октября 2023 г.
2. Songlin Sun, Jiaqi Zou, Zixuan Zou, Shaokang Wang. Experience of PYNQ. Tutorials for PYNQ-Z2. Springer Nature Singapore Pte Ltd. 2023
3. Rendering fractals on a PYNQ-Z1 [Электронный ресурс] https://github.com/lejhy/RenderingFractalsOnPYNQ-Z1#rendering-fractals-on-pynq-z1 , дата обращения 14 октября 2023 г.
4. Filip Lejhanec. Rendering Fractals Using the PYNQ Framework on an FPGA.The project's final report, University of Strathclyde, Glasgow, Scotland, 2019. https://github.com/lejhy/RenderingFractalsOnPYNQ-Z1/blob/master/Final%20Report.docx

Умножай эффективно. Алгоритм Карацубы. Прямая реализация.

Коробков Михаил, *admin@fpga-systems.ru*
Telegram @KeisN13

Обсуждение и комментарии :: ссылка

Аннотация

Операция умножения при разработке проектов на ПЛИС зачастую является базовой. В подавляющем большинстве случаев разработчики редко имеют дело с числами, разрядность которых превышает 30 (+/-). Однако в исследовательских целях, а возможно и на реальных примерах, попробуйте произвести операцию умножения для чисел большей разрядности, например 256 и посмотрите сколько ресурсов ПЛИС будет использовано, а также обратите внимание на то, как «плывут» тайминги при попытке умножить два многоразрядных числа.

В этой заметке приведен самый простой пример реализации алгоритма (реализация “в лоб”) эффективного во всех смыслах алгоритма умножения многоразрядных чисел, известный как Алгоритм Карацубы, названный так в честь автора алгоритма.

Введение

Согласно wiki: умножение Карацубы — метод быстрого умножения, который позволяет перемножать два n-значных числа с битовой вычислительной сложностью. Изобретён Анатолием Карацубой в 1960 году. Является исторически первым алгоритмом умножения, превосходящим тривиальный по асимптотической сложности.

Грубо говоря, алгоритм позволяет выполнить умножение быстрее и эффективнее, чем просто умножение в столбик.

Суть алгоритма заключается в том, что мы разделяем, но не делим, числа и проводим операции над числами меньшей разрядности.

Пример работы алгоритма

Не будем тянуть триггер за тактовый вход, а сразу перейдем к делу. Детальное описание алгоритма и как получается «магия» можно прочитать в wiki

Предположим, что мы хотим умножить два числа a = 11113333 и b = 55557777. Главный нюанс заключается в том, что числа должны быть одинаковой и четной разрядности в любой системе счисления. Если это не так, то просто добавляем слева нули и «выравниваем» их разрядности (при разборе примера в двоичной системе, далее, вы это увидите).

Шаг 1. Разделяем наши числа попалам, получаем a_left = 1111, a_right = 3333, b_left = 5555 и b_right = 7777

Шаг 2. Выполняем промежуточные вычисления:

$X = a\_left \times b\_left = 1111 \times 5555 = 6171605$

$Y = a\_right \times b\_right = 3333 \times 7777 = 25920741$

$T_0 = a\_left + a\_right = 1111 + 3333 = 4444$

$T_1 = b\_left + b\_rigth = 5555 + 7777 = 13332$

$Z = T_0 \times T_1 = 4444 \times 13332 = 59247408$

Шаг 3. Получаем результат

S – это разряды, по которым мы разделяли исходные числа. В этом примере разделения происходило по 4му разряду, то есть по 10,000 (a = 11113333 = 1111×10^4 + 3333). Это масштабирование следует учитывать при получении конечного результата

$$M = X \times S^2 + (Z - X - Y) \times S + Y = 6171605 \times 10^8 + (59247408 - 6171605 - 25920741) \times 10^4 + 2592074 = = 6171605 \times 10^8 + 27155062 \times 10^4 + 25920741 = = 617432076540741$$

Основным преимуществом Алгоритма Карацубы является то, что работа проходит с числами меньшей разрядности и в отличие от умножения в столбик, выполняется меньшее количество операций умножения. Операция масштабирования, умножение на S – это всего лишь сдвиг влево и вычислительных ресурсов на эту операцию, можно сказать, не тратится.

Также можно отметить, что Алгоритм Карацубы хорошо конвейризируется. Это означает, что интервал инициализации такой схемы при реализации на ПЛИС

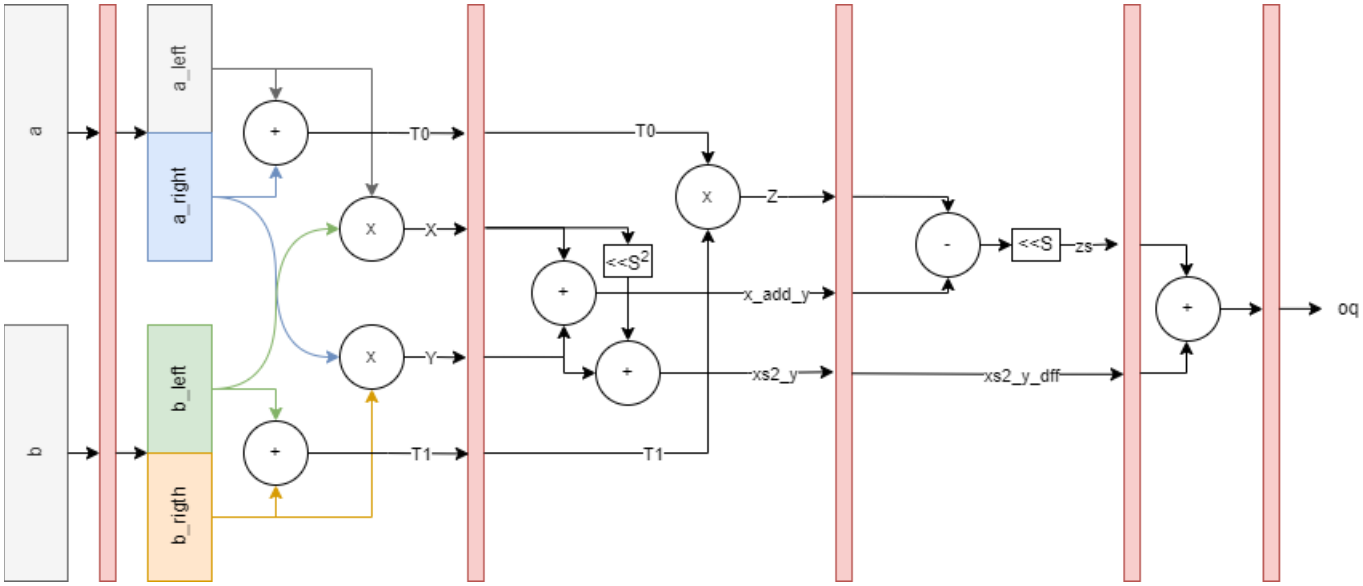


Рисунок 1

будет равным 1, то есть новые числа можно подавать каждый такт. К тому же, можно выстроить конвейер для промежуточных вычислений, что позволит увеличить частоту работы. В этом случае увеличится лишь латентность появления данных на выходе, то есть результат появится через несколько таков после начала вычислений.

Реализация

Рассмотрим реализацию Алгоритма Карацубы. Мы приведем лишь один из вариантов, который, к сожалению, не является оптимальным, а больше представляет собой реализацию “в лоб”. Другие варианты оптимизации оставим читателям в качестве домашнего задания.

Модуль будет иметь следующие порты

iclk – вход тактовой частоты
ia – первый операнд
ib – второй операнд
oq – выход, результат умножения

и всего один параметр C_WIDTH – разрядность операндов.

Следует помнить, что разрядность операндов должна быть четной и одинаковой, если это не так, просто дополняем нулями слева (листинг 1).

Следует помнить, что при умножении чисел разрядность произведения будет равна сумме разрядностей операндов, поэтому разрядность выхода

oq = C_WIDTH + C_WIDTH = 2 × C_WIDTH

```
entity Karatsuba_2 is
  generic (
      C_WIDTH : natural := 128
  );
  port (
   iclk : in std_logic;
   ia : in std_logic_vector(C_WIDTH-1 downto 0);
  ib : in std_logic_vector(C_WIDTH-1 downto 0)
   oq: out std_logic_vector(2*C_WIDTH-1 downto 0)
  );
end karatsuba;
```

Листинг 1

Перед объявлением вспомогательных сигналов, рассмотрим еще раз формулу вычисления результата, с целью понять возможность конвейеризации. Мы будем стремиться выполнять операции +, - и × параллельно и не зависимо друг от дрга за одну стадию конвейера. Структурная схема конвейера приведена на рисунке 1.

Объявим сигналы вспомогательных вычислений. В соответствии с блок-схемой алгоритма, показанной на рисунке 1.

Основное внимание здесь стоит уделить разрядности, поскольку после каждой операции она меняется.

Последним этапом необходимо выполнить построение конвейера. Для этого достаточно использовать один синхронный процесс, в котором последовательно происходит выполнение операций (листинг 3)

```
signal a_right : unsigned(C_WIDTH / 2 - 1 downto 0) := (others => '0');
signal a_left : unsigned(C_WIDTH / 2 - 1 downto 0) := (others => '0');

signal b_right : unsigned(C_WIDTH / 2 - 1 downto 0) := (others => '0');
signal b_left : unsigned(C_WIDTH / 2 - 1 downto 0) := (others => '0');

signal X : unsigned(C_WIDTH - 1 downto 0) := (others => '0');
signal Y : unsigned(C_WIDTH - 1 downto 0) := (others => '0');
signal x_add_y : unsigned(C_WIDTH downto 0) := (others => '0');

signal T0 : unsigned(C_WIDTH/2 downto 0):= (others => '0');
signal T1 : unsigned(C_WIDTH/2 downto 0):= (others => '0');

signal z : unsigned(C_WIDTH + 1 downto 0):= (others => '0');

signal xs2_y, xs2_y_dff : unsigned(X'length + C_WIDTH-1 downto 0) := (others => '0');
signal zs : unsigned(Z'length + C_WIDTH/2 - 1 downto 0):=(others => '0');

constant s : unsigned(C_WIDTH/2 - 1 downto 0) :=(others => '0');
constant s2 : unsigned(C_WIDTH - 1 downto 0) :=(others => '0');
```

Листинг 2

```
karatsuba_mult : process(iclk) begin
  if rising_edge(iclk) then
    a_right <= unsigned(ia(ia'length / 2 - 1 downto 0));
    a_left <= unsigned(ia(ia'length - 1 downto ia'length / 2));
    b_right <= unsigned(ib(ib'length / 2 - 1 downto 0));
    b_left <= unsigned(ib(ib'length - 1 downto ib'length / 2));

    X <= a_left * b_left;
    Y <= a_right * b_right;
    T0 <= ('0' & a_left) + ('0'& a_right);
    T1 <= ('0' & b_left) + ('0'& b_right);

    z <= T0 * T1;
    x_add_y <= ('0' & X) + ('0' & Y);
    xs2_y <= (x & s2 ) + Y;

    zs <= (z - x_add_y)&s;
    xs2_y_dff <= xs2_y;

    oq <= std_logic_vector(zs + xs2_y_dff);
  end if;
end process;
```

Листинг 3

Сравнение

Результаты данной реализации мягко говоря не очень. С одной стороны, она позволяет скоротить количество используемых DSP ячеек в сравнении с обычным умножением, с другой стороны обладает большим количеством уровней логики, вызванной наличием операций сложения и вычитания. Принудительное задание выполнения промежуточных вычислений на DSP секциях с использованием атрибутов приводит к увеличенному количеству используемых DSP секций и также сокращает количество уровней логики.

Предлагаю читателям самостоятельно исследовать зависимость используемых ресурсов и логических уровней от разрядности операндов и настроек синтезатора.

Upgrade

Существенным недостатком приведенной реализации является однократное разбиение входных операндов. Если рассматривать алгоритм более подробно, то существенным улучшением его реализации будет итерационное разбиение на более мелкие части промежуточных значений, произведение которых также можно выполнить по

алгоритму Карацубы. Однако, это стоит делать до определенного момента, пока это даёт хоть какой-то выигрыш по сравнению с обычным умножением, которое можно выполнить за 1 такт на DSP ячейке. Фактически нужно строить итерационную схему с длинным конвейером или использовать автомат управления, который будет последовательно передавать промежуточные значения для вычислений.

Заключение

В заметке был приведен один способов реализации Алгоритма Карацубы. Реализация не лишена недостатков, но может быть использована как стартовая точка для ее последующего улучшения и оптимизации.

Ссылки

1. Алгоритм Карацубы. Статья на wiki
2. Исходники на github
3. Запись стрима по реализации

Знаете ли вы, что первая в мире ПЛИС содержала всего 64 триггера, называлась xc2064 и была разработана в компании Xilinx, которую в то время возглавлял Ross Freeman?

На хабре есть статья про реверс инжениринг этой микросхемы, с которой мы рекомендуем ознакомиться

А еще, на этой ПЛИС совсем не давно один из участников FPGA комьюнити сделал отладочную плату

Мост сопряжения внутрикристального системного интерфейса AMBA APB4 с интерфейсом стыка простого исполнителя STI 1.0

Борисенко Николай Владимирович, *руководитель группы АО «НТЦ «Модуль», г. Москва, старший преподаватель кафедры ВТ РТУ МИРЭА.*

Телеграм: @FPGA_Mechanic
e-mail: fpga-mechanic@rambler.ru

Обсуждение и комментарии :: ссылка

Аннотация:

В статье рассмотрена организация функционального блока СБИС или ПЛИС, выполняющего функцию моста системного интерфейса, передающего транзакции чтения и записи интерфейса AMBA APB4 на инициатор стыка простого исполнителя STI версии 1.0.

Описана внутренняя организация в виде графа переходов управляющего автомата и временных диаграмм функционирования моста.

Приведена синтезируемая модель моста, описанная на языке Verilog, и тестовое окружение для верификации этой модели.

Введение

В статье предложен метод сопряжения функциональных блоков, оснащённых интерфейсом стыка простого исполнителя STI версии 1.0, с инфраструктурой системных шин AMBA фирмы ARM.

Стык простого исполнителя STI версии 1.0 описан в работах [4–12] и представляет собой синхронный системный интерфейс второго или третьего уровня иерархии в вычислительной системе, ориентированный на подключение функциональных блоков в объёме кристалла СБИС или конфигурации ПЛИС, не требующих высокой пропускной способности и возможности доступа к ОЗУ. В качестве примеров таких блоков можно привести контроллеры UART, I2C, SPI, GPIO, системные таймеры, сторожевые таймеры, управляющие системные регистры.

В семействе интерфейсов AMBA для подключения в составе интегральной схемы класса «Система на кристалле» – СнК низкоскоростных контроллеров и вспомогательных функциональных блоков применяется параллельная системная шина APB (Advanced Peripheral Bus).

Первая спецификация шины APB [1] была представлена фирмой ARM в семействе интерфейсов AMBA 2 в 1999г. и в более поздних документах была названа APB2. В спецификации APB2 был определён простейший синхронный протокол, обладавший следующими ограничениями:

1. Длительность транзакции составляла строго 2 такта, не было предусмотрено введение тактов ожидания агентом Slave.
2. Не было механизмов разрешения байтов, каждая транзакция состояла из одного обращения к 32-разрядному слову, для 32-разрядного интерфейса.
3. Разрядность шин данных записи и чтения чётко не определена.
4. Отсутствовала сигнализация ошибок.

Вторая версия спецификации шины APB [2] была опубликована в составе семейства AMBA 3 в 2004г. и называлась AMBA 3 APB Protocol Specification Version 1.0. В более поздних документах вторая версия была названа APB3. В спецификации APB3 были устранены следующие недостатки APB2:

1. Добавлен сигнал PREADY готовности агента Slave для введения тактов ожидания и увеличения длительности транзакции.
2. Добавлена сигнализация ошибок обмена с помощью сигнала PSLVERR.

Третья версия спецификации шины APB [3] была опубликована в составе семейства AMBA 4 в 2010г. и называлась AMBA APB Protocol Specification Version 2.0. Далее в этом документе третья версия была названа APB4. В спецификации APB4 были устранены следующие недостатки APB3:

1. Добавлены сигналы разрешения байтов в транзакции записи PSTRB. Обращение к одному, двум или трём байтам при чтении 32-разрядного слова осталось невозможным.

2. Добавлен сигнал PPROT, определяющий уровень доступа транзакции.

Реализация

Для моста сопряжения шины APB со стыком STI была выбрана самая совершенная спецификация APB4. Интерфейс моста сопряжения представлен на рисунке 1.

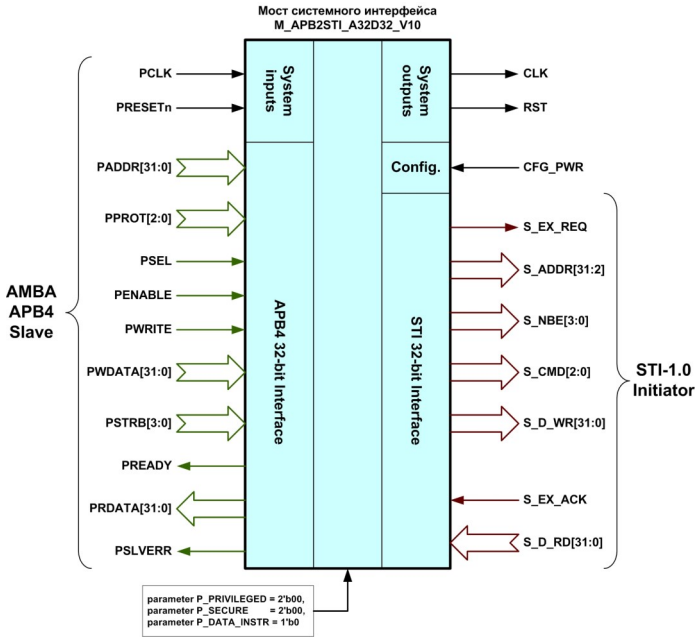


Рисунок 1 – Интерфейс моста сопряжения шины AMBA APB4 с инициатором стыка STI версии 1.0

Мост реализует передачу транзакций чтения и записи с 32-разрядной шины APB4 на 32-разрядный стык STI версии 1.0 без преобразования разрядности и без смещений слов данных в адресном пространстве.

Параметр P_PRIVILEGED имеет разрядность 2 бита и определяет реакцию моста на входной сигнал PPROT[0] (privileged/normal access) согласно таблице 1.

В случае, когда мост не транслирует транзакцию шины APB4 на стык STI, он одновременно выдаёт «1» на выходах PREADY и PSLVERR, сигнализируя ошибку доступа.

Таблица 1 – Значения параметра P_PRIVILEGED

P_PRIVILEGED[1:0]	Функционирование моста	Значение PPROT [2:0]
00	Разряд PPROT[0] игнорируется, транслируются все транзакции	XXX
01	Транслируются только транзакции «normal access», PPROT[0] = 0	XX0
10	Транслируются только транзакции «privileged access», PPROT[0] = 1	XX1
11	Разряд PPROT[0] игнорируется, транслируются все транзакции	XXX

Параметр P_SECURE имеет разрядность 2 бита и определяет реакцию моста на входной сигнал PPROT[1] (nonsecure/secure access) согласно таблице 2.

Таблица 2 – Значения параметра P_SECURE

P_SECURE[1:0]	Функционирование моста	Значение PPROT [2:0]
00	Разряд PPROT[1] игнорируется, транслируются все транзакции	XXX
01	Транслируются только транзакции «nonsecure access», PPROT[1] = 1	X1X
10	Транслируются только транзакции «secure access», PPROT[1] = 0	X0X
11	Разряд PPROT[1] игнорируется, транслируются все транзакции	XXX

Параметр P_DATA_INSTR имеет разрядность 1 бит и определяет реакцию моста на входной сигнал PPROT[2] (data/instruction access) путём трансляции транзакций чтения в адресное пространство памяти стыка STI (S_CMD = «101») или в адресное пространство памяти программ (S_CMD = «110») согласно таблице 3.

Таблица 3 – Значения параметра P_DATA_INSTR

P_DATA_INSTR[1:0]	Функционирование моста	Значение PPROT [2]
0	Все транзакции транслируются в адресное пространство памяти STI: для записи S_CMD = «0X1», для чтения S_CMD = «101».	X
1	Транзакция доступа к памяти данных APB4 транслируются в адресное пространство памяти STI: для записи S_CMD = «0X1», для чтения S_CMD = «101».	0
1	Транзакция доступа к памяти инструкций APB4 при чтении транслируются в адресное пространство памяти программ STI: для записи S_CMD = «0X1», для чтения S_CMD = «110».	1

Входной внеполосный сигнал CFG_PWR определяет порядок трансляции транзакций записи следующим образом:

- при «0» все транзакции записи транслируются с подтверждением на шине APB4 после их завершения на стыке STI;
- при «1» все транзакции записи транслируются на стык STI как отложенная запись (Posted write), без тактов ожидания, с подтверждением на шине APB4 во втором такте согласно протоколу.

Мост построен по схеме с полной регистровой развязкой от любого входа до любого выхода (за исключением выходов CLK и RST, предназначенных для синхронизации и сброса агентов стыка STI). Все выходы шины APB4 и стыка STI формируются регистрами, что исключает сквозные комбинационные цепочки и обеспечивает высокую оптимизацию быстродействия СБИС или ПЛИС.

Временная диаграмма трансляции транзакций записи показана на рисунке 2. Транзакции записи на шине APB4 выделены стрелками «APB WR», а транзакции записи на стыке STI – стрелками «STI_WR». В середине временной диаграммы режим записи изменяется путём установки высокого уровня на входе CFG_PWR, благодаря чему третья и четвёртая транзакции транслируются как отложенная запись в память.

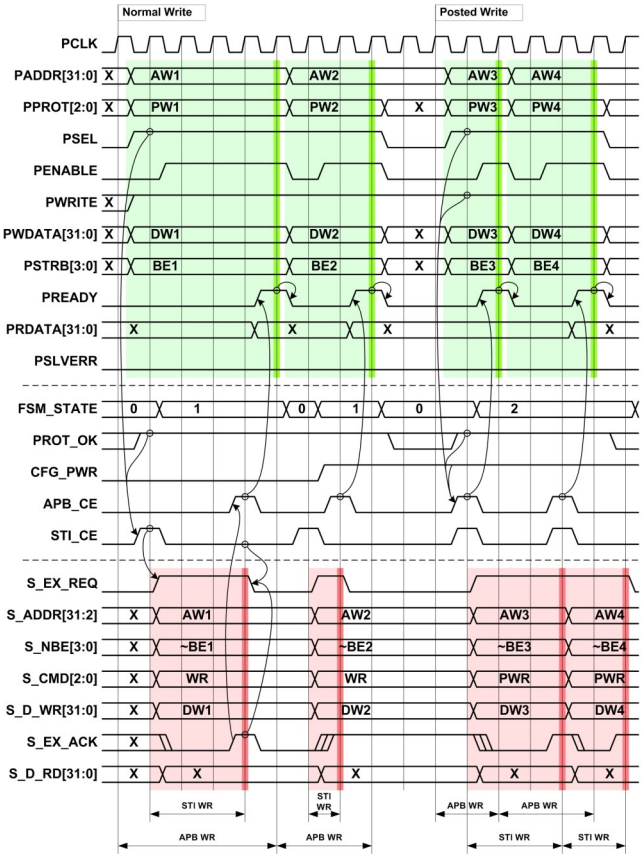


Рисунок 2 – Временная диаграмма трансляции транзакций записи

Временная диаграмма трансляции транзакций чтения показана на рисунке 3. Транзакции чтения на шине APB4 выделены стрелками «APB RD», а транзакции чтения на стыке STI – стрелками «STI_RD». Третья транзакция не транслируется на стык STI из-за ошибки доступа.

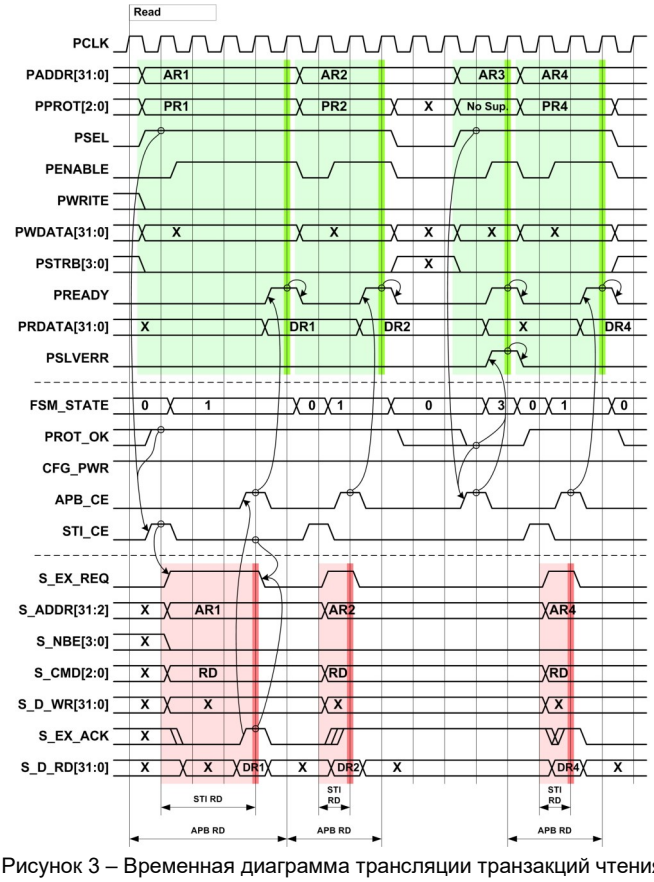


Рисунок 3 – Временная диаграмма трансляции транзакций чтения и ошибки доступа

Внутренний сигнал FSM_STATE отражает состояние управляющего конечного автомата, функционирующего согласно графу переходов, изображённому на рисунке 4.

Внутренний сигнал PROT_OK определяет действия моста согласно таблицам 1-3 следующим образом:

- при «0» транзакция не транслируется на стык STI, выдаётся сигнал ошибки доступа PSLVERR;
- при «1» транзакция транслируется на стык STI.

Внутренний сигнал APB_CE разрешает запись значений в выходные регистры шины APB4 и установку в «1» сигнала PREADY. Сброс сигнала PREADY в «0» осуществляется автоматически на следующем такте.

Внутренний сигнал STI_CE разрешает запись значений в выходные регистры стыка STI и установку в «1» сигнала S_EX_REQ. Сброс сигнала S_EX_REQ в «0» осуществляется автоматически при фиксации по фронту синхросигнала «1» на входе S_EX_ACK.

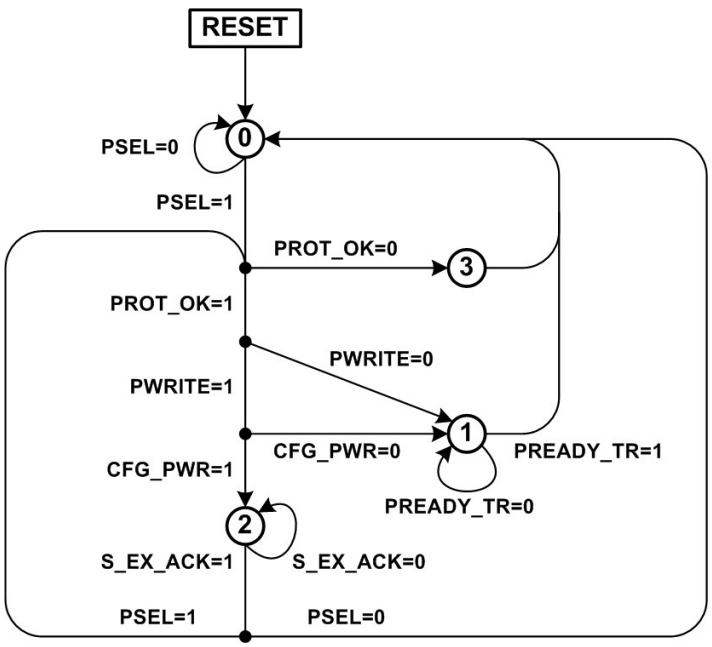


Рисунок 4 – Граф переходов управляющего конечного автомата

Синтезируемая RTL модель моста, написанная на языке Verilog, приведена ниже:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Engineer:       FPGA-Mechanic
//
// Create Date:    June.08.2017
// Design Name:    Argon SoC Proto
// Module Name:    M_APB2STI_A32D32_V10
// Target Devices: Any FPGA or ASIC
// Tool versions:  Xilinx 14.7
// Description:    AMBA 4 APB (APB4) Slave to STI (Ver.1.0) 32-bit Initiator
//                 Synchronous Bridge
// Revision:       1.0
// Revision        1.0 - File Created
//////////////////////////////////////////////////////////////////////////////////
module M_APB2STI_A32D32_V10 #(
    parameter [1:0] P_PRIVILEGED = 2'b00, // PPROT[0] Mode
    parameter [1:0] P_SECURE     = 2'b00, // PPROT[1] Mode
    parameter       P_DATA_INSTR = 1'b0)  // PPROT[2] Mode
    (
    // APB System Inputs:
    input         PCLK,    // AMBA/STI Common Clock
    input         PRESETn, // AMBA Active-Low Reset
    // STI System Outputs:
    output        CLK,     // AMBA/STI Common Clock
    output        RST,     // STI Active-High Reset
    // Write Mode Config.:
    input         CFG_PWR, // 0 - Normal write, 1 - Posted write
    // APB Slave:
    input  [31:0] PADDR,   // [1:0] are redundant, not used
    input  [2:0]  PPROT,
    input         PSEL,
    input         PENABLE, // Compatibility input, not used
    input         PWRITE,
    input  [31:0] PWDATA,
    input  [3:0]  PSTRB,
    output        PREADY,
    output reg
           [31:0] PRDATA,
    output        PSLVERR,
    // STI Initiator:
    output        S_EX_REQ,
    output reg
           [31:2] S_ADDR,
    output reg
           [3:0]  S_NBE,
    output reg
           [2:0]  S_CMD,
    output reg
           [31:0] S_D_WR,
    input         S_EX_ACK,
    input  [31:0] S_D_RD
    );

// Internal signals declaration:
 reg  S_EX_REQ_TR;
 wire STI_CE, APB_CE;
 reg  PREADY_TR, PSLVERR_TR;
 reg  PROT_OK;
 reg  [1:0] FSM_STATE;
//-----------------------------------------
// STI Output Register:
 always @ (posedge PCLK, negedge PRESETn)
  if(!PRESETn)
   begin
    S_ADDR <= 30'h00000000;
    S_NBE  <=  4'b0000;
    S_CMD  <=  3'b000;
    S_D_WR <= 32'h00000000;
   end
  else if(STI_CE)
   begin
    S_ADDR <= PADDR[31:2];
    S_NBE  <= PSTRB ^ ({4{PWRITE}});
    S_D_WR <= PWDATA;
    if(PWRITE) // Write Access
     if(CFG_PWR) // Posted Write
      S_CMD <= 3'b011; // Posted Memory Write
     else         // Normal Write
      S_CMD <= 3'b001; // Memory Write
    else        // Read Access
     if(P_DATA_INSTR) // Separate Data/Program
      if(PPROT[2]) // Instuction Access
       S_CMD <= 3'b110; // Programm Memory Read
      else // Data Access
       S_CMD <= 3'b101; // Memory Read
     else // Common Memory Mode
      S_CMD <= 3'b101; // Memory Read
   end
```

```
//-----------------------------------------
// STI Request:
 always @ (posedge PCLK, negedge PRESETn)
  if(!PRESETn) S_EX_REQ_TR <= 1'b0;
  else if(STI_CE) S_EX_REQ_TR <= 1'b1;
  else if(S_EX_ACK) S_EX_REQ_TR <= 1'b0;
 assign S_EX_REQ = S_EX_REQ_TR;
//-----------------------------------------
// APB Outputs:
 always @ (posedge PCLK, negedge PRESETn)
  if(!PRESETn) PRDATA <= 32'h00000000;
  else if(APB_CE) PRDATA <= S_D_RD;
 always @ (posedge PCLK, negedge PRESETn)
  if(!PRESETn) PREADY_TR <= 1'b0;
  else if(PREADY_TR) PREADY_TR <= 1'b0;
  else if(APB_CE)    PREADY_TR <= 1'b1;
 assign PREADY = PREADY_TR;
 always @ (posedge PCLK, negedge PRESETn)
  if(!PRESETn) PSLVERR_TR <= 1'b0;
  else if(PSLVERR_TR) PSLVERR_TR <= 1'b0;
  else if(APB_CE)     PSLVERR_TR <= ~PROT_OK;
 assign PSLVERR = PSLVERR_TR;
//-----------------------------------------
// Main FSM:
 always @ (posedge PCLK, negedge PRESETn)
  if(!PRESETn) FSM_STATE <= 2'd0;
  else
   case(FSM_STATE)

    2'd0 : // IDLE
     if(!PSEL)
      FSM_STATE <= 2'd0;
     else if(!PROT_OK)
      FSM_STATE <= 2'd3;
     else if(!PWRITE)
      FSM_STATE <= 2'd1;
     else if(!CFG_PWR)
      FSM_STATE <= 2'd1;
     else
      FSM_STATE <= 2'd2;

    2'd1 : // Normal Read/Write
     if(PREADY_TR)
      FSM_STATE <= 2'd0;
     else
      FSM_STATE <= 2'd1;

    2'd2 : // Posted Write
     if(!S_EX_ACK)
      FSM_STATE <= 2'd2;
     else if(!PSEL)
      FSM_STATE <= 2'd0;
     else if(!PROT_OK)
      FSM_STATE <= 2'd3;
     else if(!PWRITE)
      FSM_STATE <= 2'd1;
     else if(!CFG_PWR)
      FSM_STATE <= 2'd1;
     else
      FSM_STATE <= 2'd2;

    default : // Slave Error
     FSM_STATE <= 2'd0;

   endcase
//-----------------------------------------
 assign STI_CE = PSEL & PROT_OK & ((FSM_STATE == 0) |
                 ((FSM_STATE == 2) & S_EX_ACK));
 assign APB_CE = ((FSM_STATE == 1) & S_EX_ACK &
S_EX_REQ_TR) |
                 (PSEL & PROT_OK & PWRITE & CFG_PWR &
                  ((FSM_STATE == 0) | ((FSM_STATE == 2)
& S_EX_ACK))) |
                 (PSEL & ~PROT_OK &
                  ((FSM_STATE == 0) | ((FSM_STATE == 2)
& S_EX_ACK)));
//-----------------------------------------
// Protection Unit Decoder:
 always @ (PPROT)
  begin
   if(P_PRIVILEGED == 2'b01 & PPROT[0])
    PROT_OK <= 1'b0;
   else if(P_PRIVILEGED == 2'b10 & ~PPROT[0])
    PROT_OK <= 1'b0;
   else if(P_SECURE == 2'b01 & ~PPROT[1])
    PROT_OK <= 1'b0;
   else if(P_SECURE == 2'b10 & PPROT[1])
    PROT_OK <= 1'b0;
   else
    PROT_OK <= 1'b1;
  end
//-----------------------------------------
 assign CLK = PCLK;
 assign RST = ~PRESETn;
//-----------------------------------------
endmodule
```

Для верификации представленной синтезируемой модели было написано следующее тестовое окружение, воспроизводящее временные диаграммы, приведённые на рисунке 2 и рисунке 3. Тестовое окружение описано на языке Verilog следующим образом:

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////
//
// Engineer:      FPGA-Mechanic
//
// Create Date:   17:02:15 06/22/2017
// Design Name:   M_APB2STI_A32D32_V10
// Module Name:   MT_APB2STI_A32D32_V10
// Project Name:  M_APB2STI_Test
////////////////////////////////////////////////////////
//

module MT_APB2STI_A32D32_V10;
//-----------------------------------------
  reg PCLK;
  reg PRESETn;
```

```
//----------------------------------------
  reg CFG_PWR;
//----------------------------------------
  reg  [31:0] PADDR;
  reg  [2:0]  PPROT;
  reg         PSEL;
  reg         PENABLE;
  reg         PWRITE;
  reg  [31:0] PWDATA;
  reg  [3:0]  PSTRB;
  wire        PREADY;
  wire [31:0] PRDATA;
  wire        PSLVERR;
//----------------------------------------
  wire        S_EX_REQ;
  wire [31:2] S_ADDR;
  wire [3:0]  S_NBE;
  wire [2:0]  S_CMD;
  wire [31:0] S_D_WR;
  reg         S_EX_ACK;
  reg  [31:0] S_D_RD;
//----------------------------------------
   // Clock Generator - 125 MHz
   parameter PERIOD_CLK = 8; // 8ns
   parameter DUTY_CYCLE_CLK = 0.4;
   initial
    forever
     begin
      PCLK = 1'b0;
      #(PERIOD_CLK-(PERIOD_CLK*DUTY_CYCLE_CLK)) PCLK =
1'b1;
      #(PERIOD_CLK*DUTY_CYCLE_CLK);
     end
//----------------------------------------
  // Init. Reset startup pulse (100ns POR)
   initial
    begin
     PRESETn = 0;
     #100;
     @(posedge PCLK);
     #(PERIOD_CLK*0.2);
     PRESETn = 1;
    end
//----------------------------------------
  // Instantiate the Unit Under Test (UUT)
  M_APB2STI_A32D32_V10 #(
    .P_PRIVILEGED(1), // Normal access only
    .P_SECURE(0),     // DNC
    .P_DATA_INSTR(1))
    XYU (
    .PCLK(PCLK),
    .PRESETn(PRESETn),
    .CLK(),
    .RST(),
    .CFG_PWR(CFG_PWR),
    .PADDR(PADDR),
    .PPROT(PPROT),
    .PSEL(PSEL),
    .PENABLE(PENABLE),
    .PWRITE(PWRITE),
    .PWDATA(PWDATA),
    .PSTRB(PSTRB),
    .PREADY(PREADY),
    .PRDATA(PRDATA),
    .PSLVERR(PSLVERR),
    .S_EX_REQ(S_EX_REQ),
    .S_ADDR(S_ADDR),
    .S_NBE(S_NBE),
    .S_CMD(S_CMD),
    .S_D_WR(S_D_WR),
    .S_EX_ACK(S_EX_ACK),
    .S_D_RD(S_D_RD)
  );
//----------------------------------------
  initial begin
    // Initialize Inputs
    CFG_PWR = 0;
    PADDR = 32'dX;
    PPROT = 3'dX;
    PSEL = 0;
    PENABLE = 0;
    PWRITE = 1'bX;
    PWDATA = 32'dX;
    PSTRB = 4'hX;
    S_EX_ACK = 1'bX;
    S_D_RD = 32'hXXXXXXX0;

    // Wait 200 ns for global reset to finish
    #200;

    // Add stimulus here
    // WRITE:
    @(posedge PCLK);
    #(PERIOD_CLK*0.2);
    PADDR = 32'h11D9EBBA;
    PPROT = 3'd2;
    PSEL  = 1;
    PWRITE = 1;
    PWDATA = 32'h0F1E2D3C;
    PSTRB = 4'h9;
    #(PERIOD_CLK);
    PENABLE = 1;
    #(PERIOD_CLK*0.6);
    S_EX_ACK = 0;
    S_D_RD = 32'hXXXXXX1X;
    @(posedge PCLK);
    @(posedge PCLK);
    #(PERIOD_CLK*0.7);
    S_EX_ACK = 1;
    @(posedge PCLK);
    #(PERIOD_CLK*0.2);
    S_EX_ACK = 0;
    #(PERIOD_CLK);
    PADDR = 32'h11D9EBAC;
    PPROT = 3'd4;
    PWDATA = 32'h4B5A6978;
    PSTRB = 4'h8;
    PENABLE = 0;
    #(PERIOD_CLK);
    PENABLE = 1;
```

```
    CFG_PWR = 1;
    #(PERIOD_CLK*0.3);
    S_EX_ACK = 1'bX;
    S_D_RD = 32'hXXXXX2XX;
    #(PERIOD_CLK*0.3);
    S_EX_ACK = 1;
    @(posedge PCLK);
    #(PERIOD_CLK*0.2);
    #(PERIOD_CLK);
    PPROT = 3'dX;
    PSEL  = 0;
    PENABLE = 0;
    PWDATA = 32'hX;
    PSTRB = 4'hX;

    #(PERIOD_CLK*2);
    PADDR = 32'h000724BA;
    PPROT = 3'd0;
    PSEL  = 1;
    PWDATA = 32'h8796A5B4;
    PSTRB = 4'h7;
    #(PERIOD_CLK);
    PENABLE = 1;
    #(PERIOD_CLK*0.3);
    S_EX_ACK = 1'bX;
    S_D_RD = 32'hXXXX3XXX;
    #(PERIOD_CLK*0.3);
    S_EX_ACK = 0;
    @(posedge PCLK);
    #(PERIOD_CLK*0.2);
    PENABLE = 0;
    PADDR = 32'h04107F14;
    PPROT = 3'd6;
    PSEL  = 1;
    PWDATA = 32'hC3D2E1F0;
    PSTRB = 4'hB;
    #(PERIOD_CLK);
    PENABLE = 1;
    #(PERIOD_CLK*0.3);
    S_EX_ACK = 1;
    @(posedge PCLK);
    #(PERIOD_CLK*0.2);
    S_EX_ACK = 1'bX;
    S_D_RD = 32'hXXX4XXXX;
    #(PERIOD_CLK*0.3);
    S_EX_ACK = 0;
    @(posedge PCLK);
    #(PERIOD_CLK*0.2);
    PPROT = 3'dX;
    PSEL  = 0;
    PENABLE = 0;
    PWRITE = 1'bX;
    PWDATA = 32'hX;
    PSTRB = 4'hX;
    S_EX_ACK = 1;
    #(PERIOD_CLK);
    S_EX_ACK = 1'bX;

    // READ:
    #(PERIOD_CLK*10);
    PADDR = 32'h11D9EBBA;
    PPROT = 3'd6;
    PSEL  = 1;
    PWRITE = 0;
    PWDATA = 32'h00000000;
    PSTRB = 4'h0;
    #(PERIOD_CLK);
    PENABLE = 1;
    #(PERIOD_CLK*0.5);
    S_EX_ACK = 0;
    S_D_RD = 32'hXX5XXXXX;
    @(posedge PCLK);
    @(posedge PCLK);
    #(PERIOD_CLK*0.7);
    S_EX_ACK = 1;
    S_D_RD = 32'h0F1E2D3C;
    @(posedge PCLK);
    #(PERIOD_CLK*0.2);
    S_EX_ACK = 0;
    S_D_RD = 32'hX6XXXXXX;
    #(PERIOD_CLK);
    PADDR = 32'h11D9EBAC;
    PPROT = 3'd2;
    PSTRB = 4'h0;
    PENABLE = 0;
    #(PERIOD_CLK);
    PENABLE = 1;
    #(PERIOD_CLK*0.3);
    S_EX_ACK = 1'bX;
    #(PERIOD_CLK*0.3);
    S_EX_ACK = 1;
    S_D_RD = 32'h4B5A6978;
    @(posedge PCLK);
    #(PERIOD_CLK*0.3);
    S_D_RD = 32'hBABADEDA;
    @(posedge PCLK);
    #(PERIOD_CLK*0.2);
    PPROT = 3'dX;
    PSEL  = 0;
    PENABLE = 0;
    PWDATA = 32'hX;
    PSTRB = 4'hX;

    @(posedge PCLK);
    @(posedge PCLK);
    #(PERIOD_CLK*0.2);
    PADDR = 32'h000724BA;
    PPROT = 3'd1; // Privileged Access - error
    PSEL  = 1;
    PWDATA = 32'h00000000;
    PSTRB = 4'h0;
    #(PERIOD_CLK);
    PENABLE = 1;
    #(PERIOD_CLK);
    PENABLE = 0;
    PADDR = 32'h04107F14;
    PPROT = 3'd4;
    #(PERIOD_CLK);
    PENABLE = 1;
    #(PERIOD_CLK*0.2);
```

```
      S_EX_ACK = 1'bX;
      #(PERIOD_CLK*0.3);
      S_EX_ACK = 1;
      S_D_RD = 32'hC3D2E1F0;
      @(posedge PCLK);
      #(PERIOD_CLK*0.5);
      S_EX_ACK = 0;
      S_D_RD = 32'h7XXXXXXX;
      @(posedge PCLK);
      #(PERIOD_CLK*0.2);
      PPROT = 3'dX;
      PSEL  = 0;
      PENABLE = 0;
      PWRITE = 1'bX;
      PWDATA = 32'hX;
      PSTRB = 4'hX;

      #2000;
      $stop;
    end
//----------------------------------------
endmodule
```

Результат верификации в виде временной диаграммы, построенной в симуляторе Xilinx ISim, представлен на рисунке 5.

Пробная реализация моста в объёме ПЛИС XC3S400-4FG320 показала результаты по занимаемым ресурсам кристалла «Device Utilization Summary», приведённые в таблице 4.

Таблица 4 – Ресурсы ПЛИС XC3S400, занятые мостом APB4-STI

Ресурс	Задейст-вовано	Доступно	Расход
Slice Flip Flops	4	7,168	1%
occupied Slices	12	3,584	1%
4 input LUTs	16	7,168	1%
bonded IOBs	210	221	95%
IOB Flip Flops	101		
BUFGMUXs	1	8	12%

Допустимый период тактового сигнала PCLK после реализации моста в кристалле ПЛИС XC3S400 согласно отчёту «Place and Route» САПР Xilinx ISE составил 6,433нс, что соответствует максимальной тактовой частоте 155 МГц.

Выводы

Предложенная модель моста интерфейса AMBA APB4 позволяет сопрягать функциональные блоки для стыка простого исполнителя STI версии 1.0 с инфраструктурой интерфейсов AMBA и процессорными ядрами фирмы ARM. Модель использует минимальное количество ресурсов кристалла и имеет регистровую развязку, обеспечивающую высокую тактовую частоту.

Мост поддерживает все расширения интерфейса AMBA APB, включая уровни доступа, разрешения байтов при записи и сигнализацию ошибок.

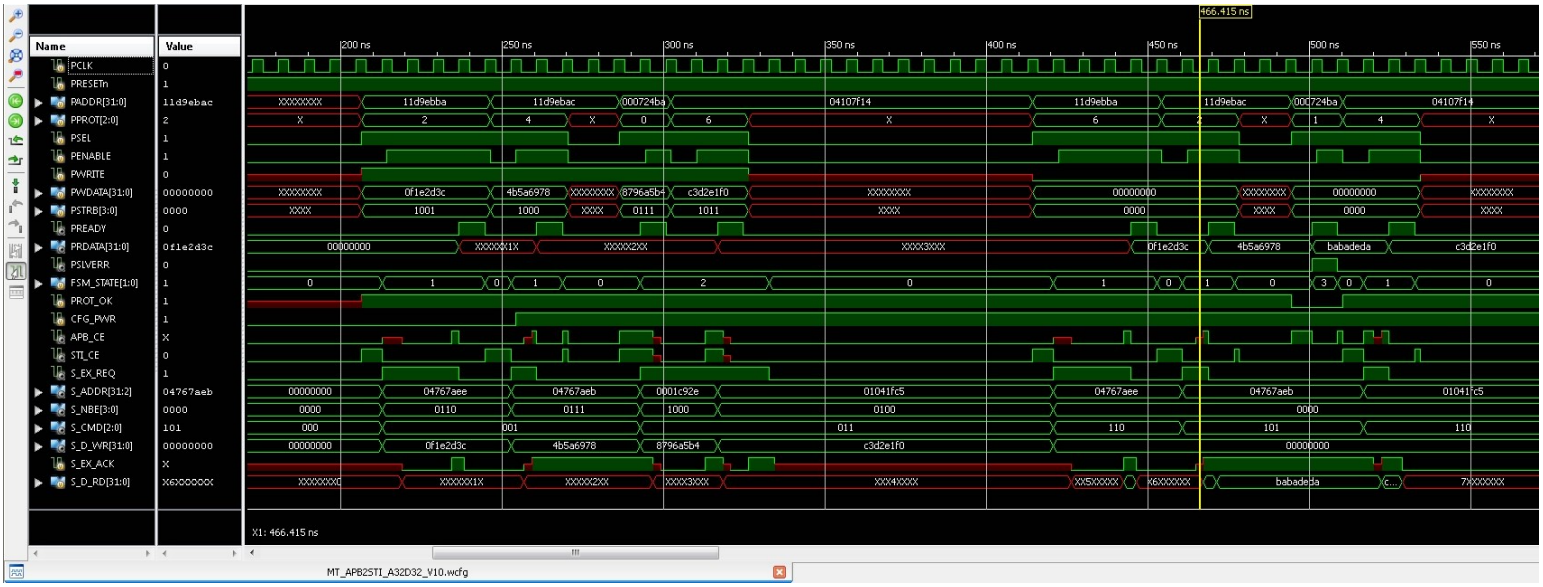


Рисунок 5 – Результат верификации в симуляторе Xilinx ISim

Транзакции записи могут транслироваться мостом в режиме отложенной записи, что уменьшает задержку одиночной транзакции записи на шине AMBA APB4.

Описанная модель моста была успешно апробирована в проекте СнК Xilinx Zynq Z-7010. Сбоев в работе шины APB4 и стыка STI не выявлено.

Список источников

1. AMBA Specification Rev 2.0 ARM IHI 0011A 1999, ARM.
2. AMBA 3 APB Protocol. Version: 1.0. Specification. ARM IHI 0024B 2003, 2004, ARM.
3. AMBA APB Protocol. Version: 2.0. Specification. ARM IHI 0024C (ID041610) 2003-2010, ARM.
4. Борисенко Н. В. Синхронный системный интерфейс взаимодействия вычислительных ядер с периферийными блоками кристалла СБИС. «Компоненты и технологии» №10.2016.
5. https://habr.com/ru/post/354818/
6. Борисенко Н. В. Организация коммутирующей шинной инфраструктуры, соединяющей агенты синхронного системного интерфейса Simple Target Interface - STI версии 1.0. «Компоненты и технологии» №5.2017.
7. https://habr.com/ru/post/354852/
8. Фролова Е. Д. Организация синтезируемых моделей блоков статической памяти для системного интерфейса STI. «Компоненты и технологии» №6.2017.
9. Борисенко Н. В. Организация сопряжения внутрикристального синхронного ОЗУ с синхронной системной шиной без конвейеризации адресов и без команд чтения с предвыборкой. «Компоненты и технологии» №12.2020.
10. Борисенко Н. В. Синтезируемая модель арбитра доступа к среде передачи данных. «Компоненты и технологии» №8.2011.
11. Борисенко Н. В. Набор унифицированных протоколов для организации синхронного взаимодействия функциональных узлов и блоков цифровой аппаратуры. Часть 1 и часть 2. «Компоненты и технологии» №08.2018, №09.2018.
12. Борисенко Н. В. Организация автоматического коммутатора двухстороннего пакетного обмена словами данных фиксированной разрядности в объёме кристалла СБИС или ПЛИС. «Компоненты и технологии» №6.2022.

Многоканальное устройство записи (МУЗА_4К10М1)

Сухачев К.И., Шестаков Д.А.
e-mail: kir.sukhachev@gmail.com

Обсуждение и комментарии :: ссылка

При проведении различных опытов и экспериментов [1] существует необходимость иметь встраиваемый модуль, позволяющий вести запись аналоговых сигналов, при этом иметь уникальные функции обработки, детектирования и синхронизации недоступные готовым изделиям. В связи с этим, было принято решение разработать несложное устройство с базовым функционалом осциллографической приставки и возможностью внесения изменений как в структуру записываемых каналов, так и в последующую обработку принятых данных. Внешний вид разработанного модуля (МУЗА_4К10М1) показан на рисунке 1.

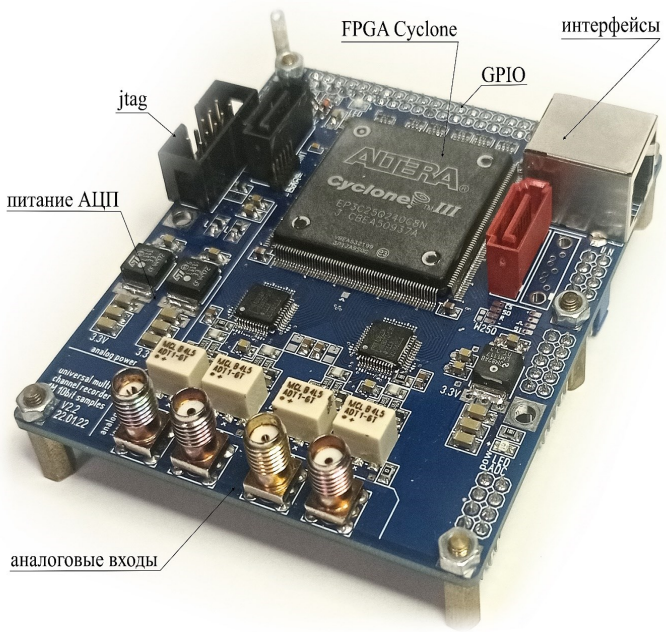


Рисунок 1 – Многоканальное устройство записи (МУЗА_4К10М1)

Послойная топология модуля показана на рисунке 2. На плате кроме FPGA EP3C25Q240C8 или EP3C16Q240C8 и двух АЦП AD9218 или AD9288 расположены все необходимые цепи питания, разные для цифровых и аналоговых частей, защиты по питанию, входные согласующие трансформаторы, два ОЗУ IS61WV102416 с общей шиной адреса. На плате присутствуют стандартные разъемы, (разъемы, а не полноценные интерфейсы!!!) позволяющие использовать стандартные, заводские кабели, например для каскадирования нескольких таких модулей. Свободные выводы FPGA выведены на периферийные разъемы, что позволяет существенно расширять функционал модуля, например, превратив его в высокоскоростную систему управления какими-то процессами. Плата выполнена в формате 4-слойной, внутренние слои отведены преимущественно под линии питания и опорный полигон, который в принципе присутствует в качестве заливки и на остальных слоях. Опорный полигон выполнен с пространственным разделением аналоговой и цифровых «земель» с соединением в областях АЦП.

Данный модуль предполагался для работы при пониженном давлении, поэтому было определено распределение тепловых полей в установившемся режиме (рисунок 3), согласно которым проработана система отведения тепла от модуля в условиях отсутствия конвекции.

Особенности

- Доступно от 1 до 4 каналов с частотой дискретизации до 100 МГц, 3 канала полные:10 -бит и 1 канал поступает только на компаратор (эквивалент канала логического анализатора);
- Глубина памяти до миллиона выборок при записи 4х каналов с возможностью регулировки;
- Возможность подключения 8-битных и 10-битных АЦП;
- Запись предыстории с возможностью регулировки;
- Установка уровня срабатывания компаратора на любой из каналов
- Настраиваемая фильтрация;
- Режимы работы: однократный, нормальный и с внешней синхронизацией;
- Встроенные математические функции;

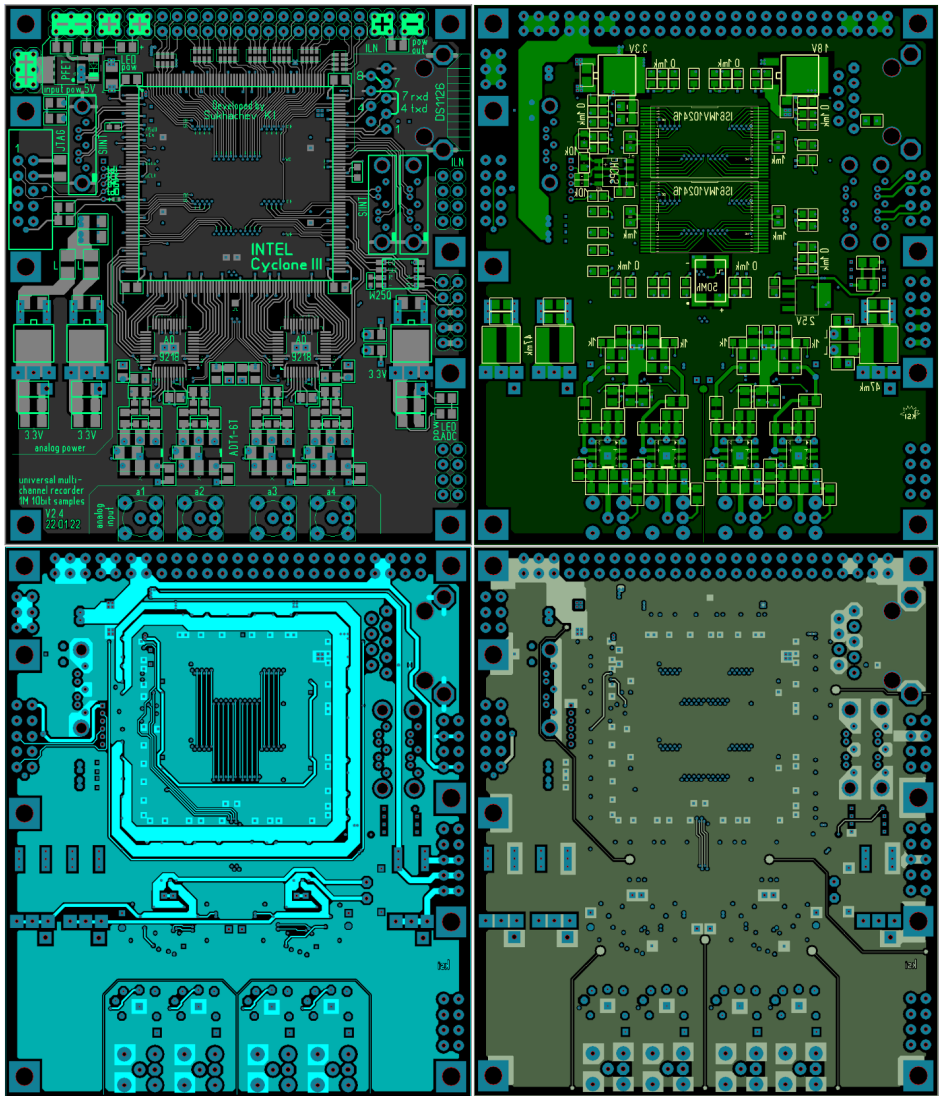


Рисунок 2 – Послойная топология модуля (МУЗА_4К10М1) для 4-слойной ПП со сквозной металлизацией ПО. Размеры модуля 78Х91мм.

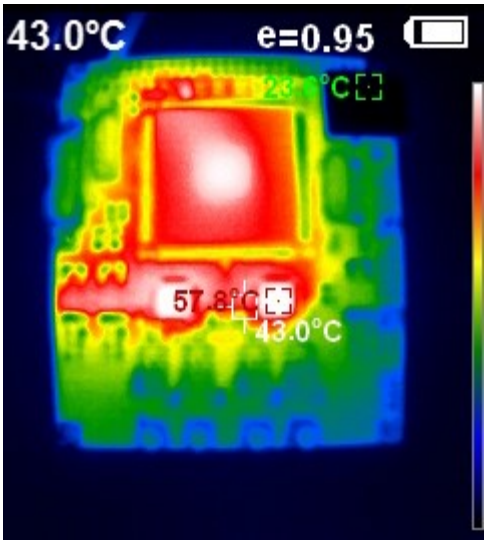


Рисунок 3 – Температура модуля (МУЗА_4К10М1) в установившемся режиме

- Доступные (в актуальной версии) интерфейсы: UART, QSPI/SPI, SINT.

Описание

Микросхема управления представляет собой FPGA с объемом 25000 логических ячеек и банком встроенной памяти. Для работы модуля необходимо подключение внешних ИМС АЦП и статической ОЗУ. Внешние микросхемы могут быть выбраны исходя из целевых требований к конечному проекту, в данной статье представлены только апробированные характеристики.

Управление и взаимодействие с внешними устройствами или ПК осуществляется через один из доступных интерфейсов. Основным протоколом является SINT [2], однако UART прослушивается в фоновом режиме и если основной – «молчит», то команды с UART проходят и обрабатываются. На рисунке 4 показана структура верхнего уровня описания для ПЛИС.

Функционирование

Устройство «МУЗА4К10М1» при наличии разрешающего флага «Работа» циклически записывает во внешнюю статическую оперативную память (ОЗУ) три 10ти битных канала с АЦП. Четвертый канал записывается в формате одного бита, который свидетельствует о детектировании события в канале по заданным параметрам. Последний свободный бит в шине данных внешнего ОЗУ – это бит события ведущего канала, т.е. канала, по которому формируется условие прекращения циклического процесса записи в ОЗУ с последующей записью «постистории» и генерацией внутреннего флага готовности к считыванию данных внешним устройством по активному интерфейсу. В качестве ведущего канала может быть выбран любой из четырех возможных. Однако четвертый канал не может быть записан в ОЗУ в полном формате, а только один бит, детектирующий событие в нем.

Детектирование событий по каналам может происходить по различным сценариям: например, превышение жестко заданного порога (осциллографический режим) или адаптивное распознавание формы импульса (адаптивный режим).

Адаптивный режим может содержать различные параметры, однако в актуальной версии он предназначен для детектирования импульса двухполярного

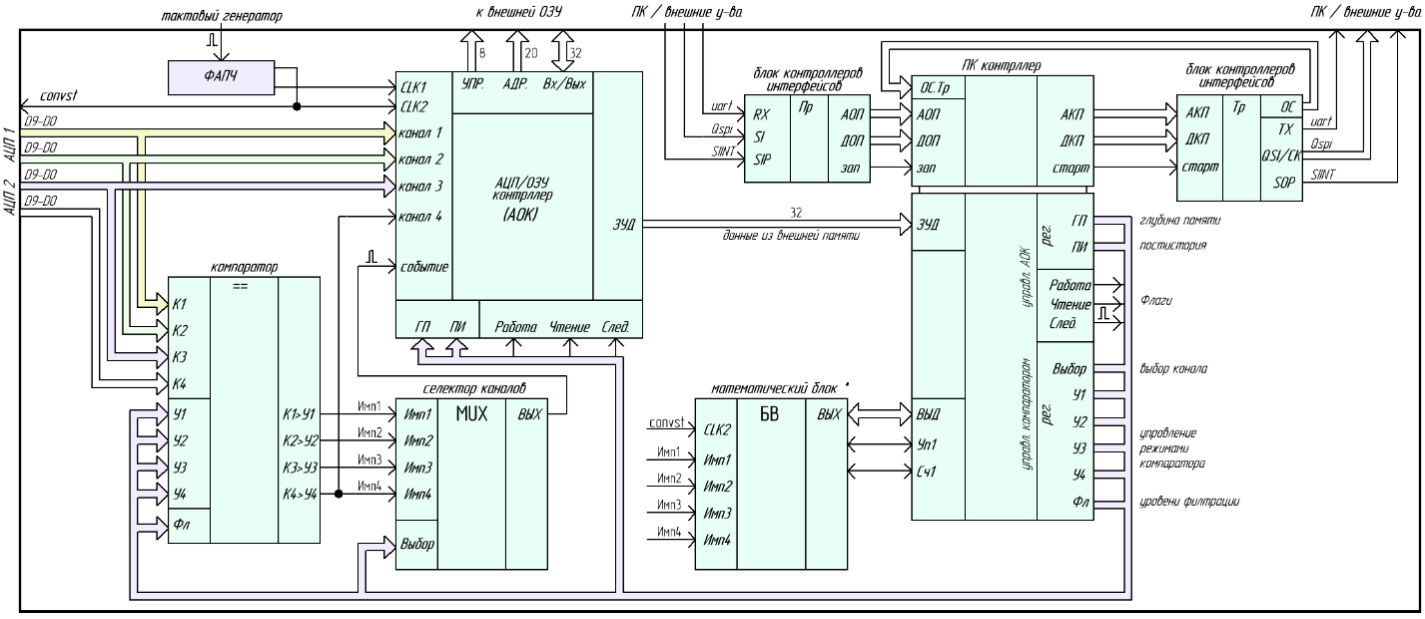


Рисунок 4 – Структура соединения модулей в файле верхнего уровня

экспоненциального импульса. Иллюстрация процесса записи кадра представлена на рисунке 5.

Пример одного из сценариев работы:

- Задаются значения управляющих регистров (или остаются значения «по умолчанию»). Доступно задание:
  - глубины памяти;
  - объема пост истории в рамках заданной глубины памяти;
  - ведущего канала;
  - параметров фильтрации и детектирования события;
  - режимов функционирования.
- Посылается команда «однократной записи», при этом в автоматическом режиме происходит фильтрация сигнала и детектирование события с заданными параметрами по выбранному каналу в качестве ведущего;
- Происходит запись кадра и постистории в заданном объеме;
- Далее осуществляется отправка на ведущее устройство (например ПК) пакета данных, содержащего заданный объем записанной с АЦП информации. Отправка идет в непрерывном режиме в рамках заданного объема глубины записи кадра.
- Устройство «МУЗА4К10М1» будет находиться в режиме ожидания: все 4 канала будут закрыты. Для дальнейшей работы необходима повторная отправка команды «однократной записи» или принудительный переход в другой режим, например, через команды: «сброс текущего процесса», «полный сброс» или

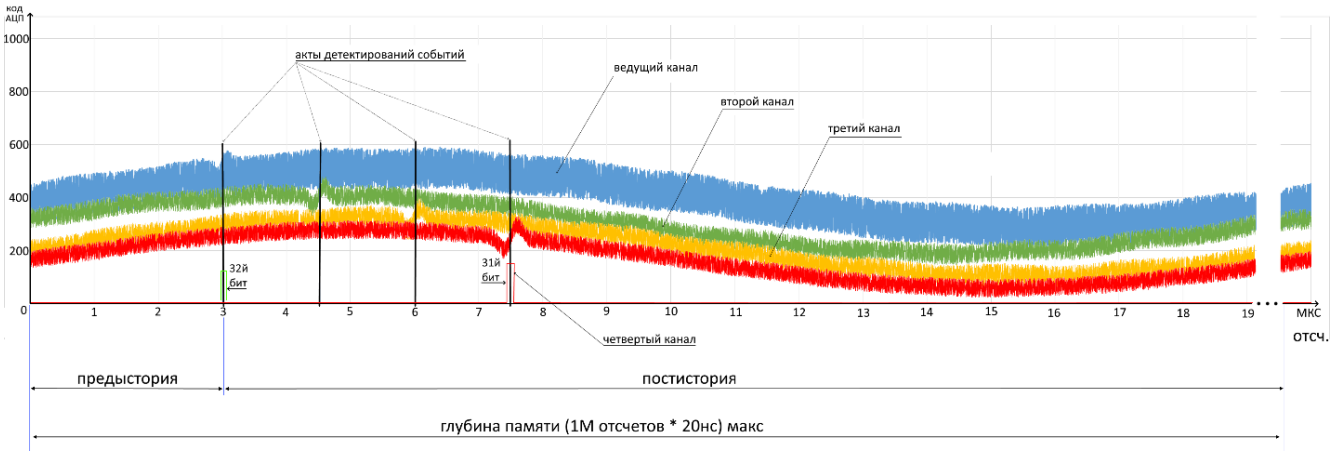


Рисунок 5 – Процесс записи кадра в однократном или нормальном режимах

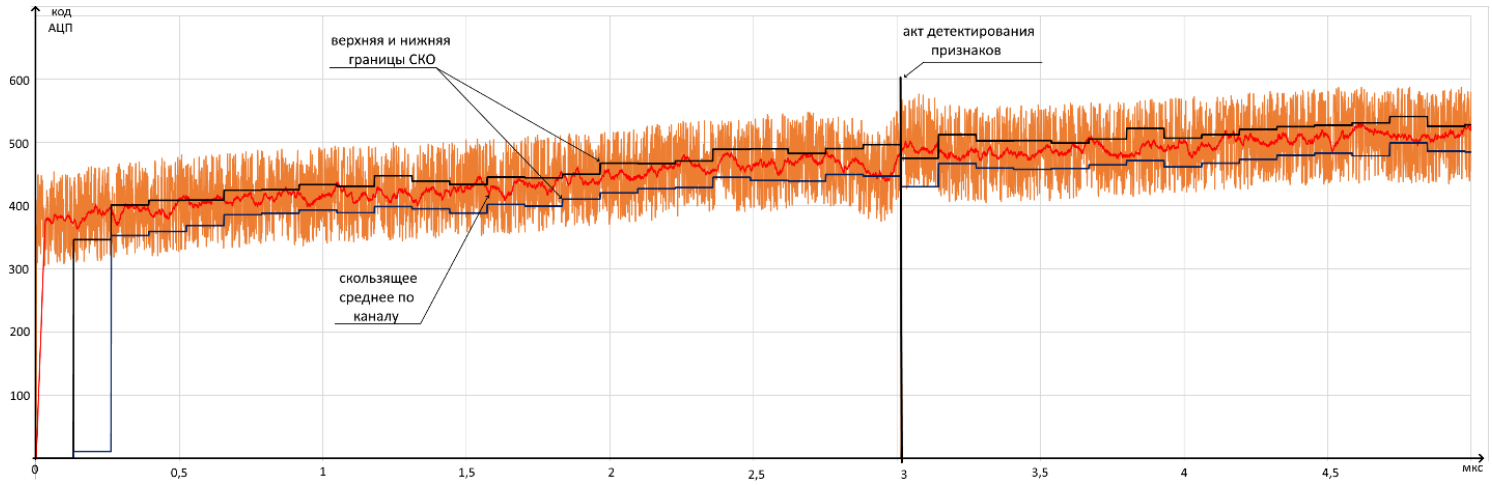


Рисунок 6 – Процесс записи кадра в однократном или нормальном режимах (фильтрация и экстраполяция по СКО)

задание другого режима: «нормальный режим» или «автоматический».

- В режиме ожидания доступно повторное считывание данных из ОЗУ по команде «повторное чтение памяти».

Система детектирования

Система детектирования по уровню срабатывает при непрерывном превышении заданного порога в течении определенного времени (т.е. кол-ва отсчётов АЦП).

Адаптивная система производит фильтрацию по скользящему среднему в формате (8, 16, 32, 64 или 128), а также вычисление СКО [3-5] от полученного усредненного значения с его экстраполяцией на следующий «псевдостационарный» период, в границах которого считается, что НЧ составляющая вносит не существенный вклад в изменение усредненного значения с АЦП (рисунок 6). Величина данного периода задается от внешнего устройства или адаптируется самостоятельно. При превышении усредненного значения с ведущего канала АЦП адаптивного порога, полученного от экстраполяции СКО с учетом поправочных коэффициентов, происходит формирование признаков искомого сигнала. При совпадении всех признаков формируется флаг детектирования события по заданному каналу, останавливается циклическая запись, дописывается постистория и формируется флаг готовности считывания ОЗУ. Один из вариантов описания компаратора приведен в приложении 1.

Управление

Для управления необходимо отправить в подряд 4 байта по UART на скорости до 2Мбит. Первые два байта формируют регистр управления ATS, следующие два байта из четырехбайтного пакета формируют регистр DTS. Для формирования любой команды необходимо обновить содержимое ATS и DTS Команды можно отправлять в подряд без разделительных интервалов. Если отправка 4-байтного пакета нарушена и отправлено меньшее число байт, то команда сформирована не будет, и после закрытия временного окна ожидания процесс приема пакета будет остановлен. Формирование управляющих команды и задание значений для внутренних настроечных регистров, происходит одинаково - через механизм обновления регистров ATS и DTS, структура 4-байтного управляющего пакета показана на рисунке 7.

В отличие от UART интерфейс SINT [2] напрямую формирует регистры ATS и DTS за одну посылку.

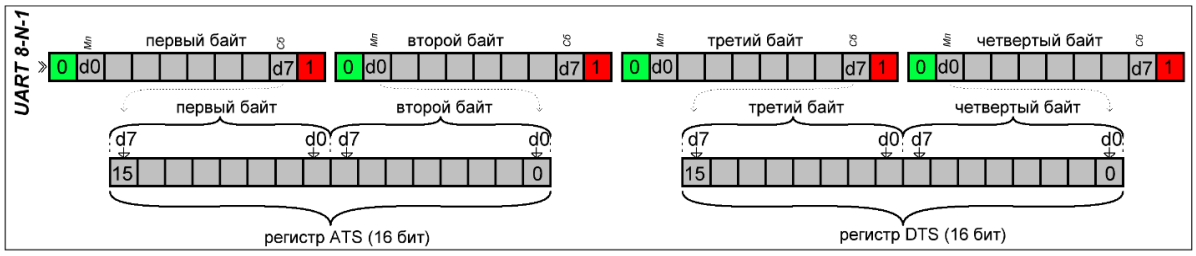


Рисунок 7- Формат управляющего пакета

Команды задают режимы работы, т.е., требуют включения режима или его отключения, либо вызывают однократную операцию, например, ответную отправку содержимого внешней памяти или содержания управляющих регистров. Коды команд задаются согласно таблице 1.

Управляющие регистры имеют установки «по умолчанию», однако они могут быть переопределены, если по определенным адресам (регистр ADR) отправить новые значения (регистр DTR). Адреса управляющих регистров задаются согласно таблице 2. Существуют специальные команды, переназначающие основной канал управления или организующие сброс системы управления или полный сброс внутренних регистров. Список специальных команд представлен в таблице 3.

Вывод

Представлен модуль, позволяющий вести запись аналоговых сигналов по 4м каналам (три 10-битных аналоговых и один логический, или 4 аналоговых 8-битных). Существует гибкая возможность при сохранении базовых HDL модулей через добавление модуля обработки добиться необходимых характеристик и возможностей системы. Разработаны и испытаны HDL описания базовых цифровых модулей, испытания которых проводились в том числе на записи реального сигнала с целевых датчиков и усилителей.

Источники

1. K. I. Sukhachev. Operating Algorithm of the Digital Module of the Device for Detecting Flight Pulse// Sukhachev, Telegin, Grigoriev, Shestakov, Dorofeev -Instruments and Experimental Techniques 66, pages228–233 (2023).
2. https://habr.com/ru/articles/769986/
3. E. Thomas, S. Auer, K. Drake, M.Horányi,T.Munsat, A. Shu//Planetary and Space Science. 2013. Vol. 89. P.71–76. Doi 10.1016/j.pss.2013.09.004.
4. Brakel, J.P.G. Robust peak detection algorithm using z-scores // Stack Overflow. – 2014. – URL: https://stackoverflow.com/questions/22583391/peak-signal-detection-in-realtime-timeseries-data/22640362#22640362 (Дата обращения: 20.05.2022).
5. Каламбет Ю.А., Мальцев С.А., Козьмин Ю.П. // Заводская Лаборатория. Диагностика Материалов. 2015. 81. С. 69–76.

Таблица 1 – коды команд и флагов

ADR (16 бит)	DTR (16 бит)	Описание
Dec: 1000	Dec: 1	Установка флага «работа» в активное состояние
	Dec: 2	Установка флага «работа» в неактивное состояние
	Dec: 3	Установка флага «чтение» в активное состояние
	Dec: 4	Установка флага «чтение» в неактивное состояние
	Dec: 5	Установка флага «след» в активное состояние
	Dec: 6	Установка флага «след» в неактивное состояние
	Dec: 10	Комплексная команда «Однократный режим»
	Dec: 15	Комплексная команда переход в «Нормальный режим»
	Dec: 16	Комплексная команда выход из «Нормального режима»
	Dec: 300	Комплексная команда выход из «Автоматический режим»
	Dec: 301	Комплексная команда выход из «Автоматического режима»

Таблица 2 – адреса управляющих регистров и операции с ними

ADR (16 бит)	DTR (16 бит)	Описание
Dec: 101	Dec: X1	запись в старший 16-битный сектор регистра «ГП» числа X1
Dec: 102	Dec: X2	запись в младший 16-битный сектор регистра «ГП» числа X2
Dec: 111	Dec: X1	запись в старший 16-битный сектор регистра «ПИ» числа X1
Dec: 112	Dec: X2	запись в младший 16-битный сектор регистра «ПИ» числа X2
Dec: 121	Dec: Y1+ Y2	Запись в 16-битный регистр «У1» двух 8-битных чисел Y1 и Y2
Dec: 122	Dec: Y1+ Y2	Запись в16-битный регистр «У2» двух 8-битных чисел Y1 и Y2
Dec: 123	Dec: Y1+ Y2	Запись в16-битный регистр «У3» двух 8-битных чисел Y1 и Y2
Dec: 124	Dec: Y1+ Y2	Запись в 16-битный регистр «У4» двух 8-битных чисел Y1 и Y2
Dec: 125	Dec: Y1+ Y2	Запись в 8-битные регистры «Выбор» и «ФЛ» двух 8-битных чисел Y1 и Y2

Таблица 3 – коды специальных команд

ADR (16 бит)	DTR (16 бит)	Описание
Dec: 1000	Dec: 60000	Отключение функции «эхо»
	Dec: 60001	Включение функции «эхо»
	Dec: 60005	Переход на интерфейс «SINT»
	Dec: 60006	Переход на интерфейс «UART»
	Dec: 60050	Пороговый режим компаратора
	Dec: 60051	Адаптивный режим компаратора
	Dec: 60100	Команда сброса
	Dec: 60101	Команда запроса статуса

Приложение – 1

фрагмент описания модуля цифрового детектора

```
module adaptiv_SCPM // adaptive tuned detector module for tasks of the accelerator
(CLK, reset, run,          // main input signal    (necessary)
ADC_1, ADC_2,         // codes from two ADCs  (necessary)
average_ADC,          // Average output data  (optional)
// externally configurable moving average and adaptive detector parameters:
filter, static_time, shift, mult, delay,   singA1_limit, singB1_limit, singC1_limit,
event_1ch, event_2ch);     // event detection signal
input wire [5:0] filter;   // externally parameter for "moving average"
// externally parameter "adaptiv":
input wire [7:0] static_time;
input wire [7:0] singA1_limit; input wire [7:0] singB1_limit; input wire [7:0] singC1_limit;
input wire [2:0] shift;
input wire [5:0] mult;
input wire [9:0] delay;
input wire CLK, reset, run;  // main input signal
input wire [9:0] ADC_1;  input wire [9:0] ADC_2;  // codes from two ADCs
output reg event_1ch, event_2ch;  // characteristic signal detection output
// internal registers "moving average" module:
integer i, j;
reg [17:0] MA_adder;
wire [9:0] ADC; assign ADC[9:0]=select_chl?ADC_2:ADC_1;
output reg [9:0] average_ADC;
reg [9:0] average_buf [64:0];
// internal registers "adaptiv mode" module:
reg select_chl;
reg [9:0] maximus_MA1;  reg [9:0] minimus_MA1;
reg [9:0] mem_max;  reg [9:0] mem_min;  reg [9:0] counter_sing;
reg [9:0] counter_sing2;
reg sing_A1, sing_B1, sing_C1;  //signs of a characteristic signal
// other internal registers:
reg [7:0] static_counter;
reg [20:0] delayer;
// moving average, MA adder:
wire [3:0] average_LVL;
wire [6:0] filter_M; assign filter_M[6:0]=filter[5:0]+1'd1;
assign average_LVL[3:0]  =   filter_M[6]?4'd6:   filter_M[5]?4'd5:
              filter_M[4]?4'd4:
              filter_M[3]?4'd3:
              filter_M[2]?4'd2:
              filter_M[1]?4'd1:
              4'd0;
always @ (posedge CLK or posedge reset) begin
if (reset) begin
  MA_adder<=0; maximus_MA1<=0; minimus_MA1<=10'd1023;
  mem_max<=0; mem_min<=0; counter_sing<=0; counter_sing2<=0;
  sing_A1<=0; sing_B1<=0; sing_C1<=0;
  static_counter<=0; average_ADC<=0; select_chl<=0; delayer<=0;
```

```
  event_1ch<=0; event_2ch<=0;
end // reset
else begin
if (run) begin
  average_buf[0] <= ADC;
  MA_adder<=MA_adder+average_buf[0]-average_buf[filter_M];
  for (j=0; j<=filter; j=j+1) average_buf[j+1] <=average_buf[j];    average_ADC[9:0]<=MA_adder>>average_LVL;
if (static_counter==static_time) begin
  static_counter<=0;
  mem_max[9:0]<=maximus_MA1[9:0]+(mem_max[9:0]>>mult);
maximus_MA1<=0; minimus_MA1<=10'd1023;
mem_min[9:0]<=minimus_MA1[9:0]-(mem_min[9:0]>>mult);
end // static_time
else begin
  static_counter<=static_counter+1'd1;
  if (maximus_MA1<average_ADC) maximus_MA1[9:0]<=average_ADC[9:0];
  if (minimus_MA1>average_ADC) minimus_MA1[9:0]<=average_ADC[9:0];
  if (!sing_C1) begin
    if (!sing_B1) begin
      if (average_ADC<mem_min) begin
        if (!sing_A1) begin
        if (counter_sing<singA1_limit) counter_sing<=counter_sing+1'd1;
        else begin
          sing_A1<=1'd1; counter_sing<=counter_sing<<shift;
        end
        end // sing_A1
      end // average_ADC<mem_min
      else begin
        if (sing_A1) begin
          if (counter_sing==0)   sing_A1<=0;
          else begin
            if (average_ADC>mem_max) begin
              if (counter_sing2==singB1_limit) begin
                sing_B1<=1'd1;
                counter_sing<=counter_sing2;
                counter_sing2<=0;
              end
            counter_sing2<=counter_sing2+1'd1;
            end
          else begin
          counter_sing<=counter_sing-1'd1;
          counter_sing2<=0;
          end
        end
    end // sing_A1 = 1 & average_ADC> mem_min
    else counter_sing<=0;
  end // average_ADC>mem_min
end // !sing_B1
  else begin
```

```
if (average_ADC>mem_max) begin
  if (counter_sing==singC1_limit) begin
    sing_C1<=1'd1;
    counter_sing<=0;
    if (select_chl) event_2ch<=1'd1;
    else   event_1ch<=1'd1;
  end
else counter_sing<=counter_sing+1'd1;
end // average_ADC>mem_max
else begin
if (counter_sing==0) begin
  sing_A1<=0;  sing_B1<=0;
  sing_C1<=0;
  counter_sing<=0;
end
else counter_sing<=counter_sing-1'd1;
end // average_ADC<mem_max
end // sing_B1
end // !sing_C1
else begin
  if (counter_sing==delay) begin
    sing_A1<=0;  sing_B1<=0; sing_C1<=0;
    counter_sing<=0; delayer<=0;
  end
  else begin
    counter_sing<=counter_sing+1'd1;
    if (counter_sing==10'd10) begin
      if (select_chl) event_2ch<=1'd1;
      else   event_1ch<=1'd1;
      delayer<=0;
    end
  end
  if (delayer!=21'd1_000_000) delayer<=delayer+1'd1;
  else begin
    sing_A1<=0;  sing_B1<=0; sing_C1<=0;
    counter_sing<=0; delayer<=0;
  end
 end // sing_C1
 if (event_1ch==1'd1) event_1ch<=0; // event detection clock
 if (event_2ch==1'd1) event_2ch<=0; // event detection clock
 end // static_time
 end // run
 end // CLK
end // always
endmodule // adaptiv_SCPM
```

Реализация передатчика MIPI CSI-2 на GOWIN GW2A с подключением к Raspberry PI

Минаев Александр, *aleksandr.minaiev@gmail.com*,
Telegram @aleks_andr_m

Обсуждение и комментарии :: ссылка

1. Описание проекта и используемое оборудование

В рамках работы над проектом стерео-камеры было решено использовать для передачи изображения интерфейс MIPI CSI-2. Данный интерфейс был выбран по нескольким причинам:

1. Высокая скорость передачи
2. Распространенность
3. Возможность в дальнейшем подключать несколько камер без нагрузки на процессор

Идея использовать данный интерфейс появилась после попытки одновременного подключения 6 стереокамер к Jetson TX2. После третьей камеры драйвер UVC выдал ошибку "libv4l2: error turning on stream: No space left on device", т.к. интерфейсу USB не хватало запрашиваемой полосы пропускания.

В связи с санкциями было решено использовать ПЛИС из Китая. Выбор пал на плату Sipeed Tang Primer 20K – SoM-модуль в формате SO-DIMM (Рисунок 1).

Рисунок 1 – Внешний вид платы Sipeed Tang Primer 20K

Параметры модуля, интересные в рамках проекта:

- 20736 LUT,
- 828 Kbit Block SRAM,
- достаточное количество выводов, в том числе выровненных по длине LVDS пар
- 1Gbit (128MB) DDR3
- Доступность и дешевизна
- Бесплатные IP ядра для Standart версии IDE

Была разработана несущая плата с двумя разъемами DVP для камер OV5640 и SCI-2 интерфейсом.

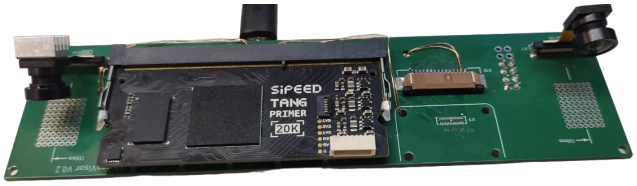


Рисунок 2 – Плата стереокамеры (с ошибками)

Для подачи правильного напряжения питания на банки 0, 1 и 7 ПЛИС с модуля были выпаяны резисторы R5 и R9

2. Описание интерфейса SCI-2

Поскольку стояла задача подключения устройства к встраиваемым системам, было решено повторить распиновку разъема камеры Raspberry Pi V2.1 (Рисунок 3).

Физический уровень интерфейса реализован в соответствии со спецификацией D-PHY. Каждая дифференциальная пара интерфейса работает в двух режимах: Low Power (LP) и High Speed (HS) (Рисунок 4). Таким образом одна пара интерфейса занимает четыре вывода ПЛИС: два True-LVDS (напряжение банка 2.5 В) и два LVCMOS-1.2V (напряжение банка 1.2 В).

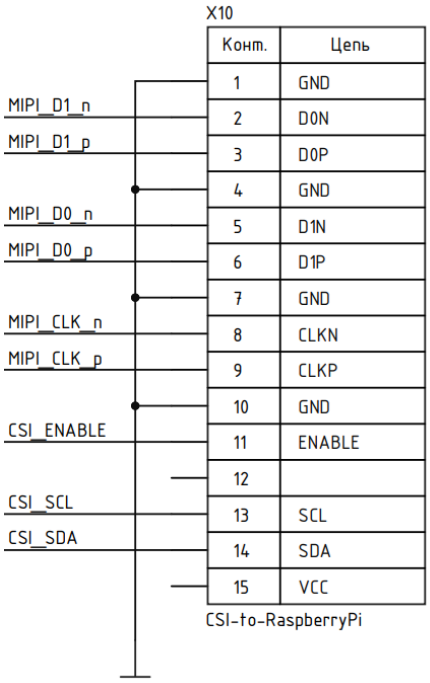


Рисунок 3 – Разъем CSI-2

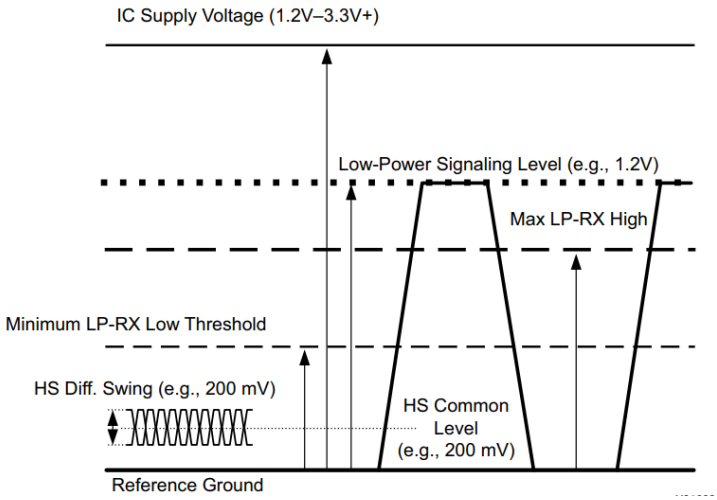


Рисунок 4 – Уровни HS и LP

LP выводы подключаются к HS выводам через резисторы номиналом 100 Ом.

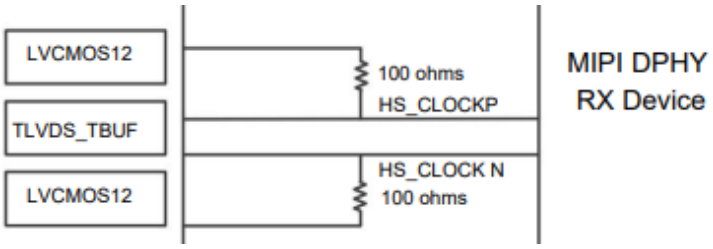


Рисунок 5 – Соединение выводов

В неактивном режиме интерфейс находится в состоянии LP11, на обоих выводах присутствует высокий уровень 1.2 В. В начале передачи состояние линий переходит в 01, далее в 00. После этого LP выводы переключаются в состояние High-Z и начинается передача данных в режиме HS (Рисунок 6).

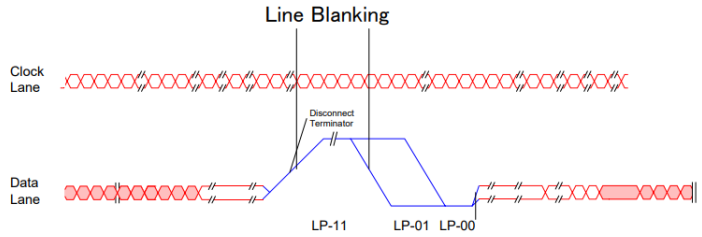


Рисунок 6 – Начало передачи данных

Структура пакета показана на рисунке 7. Существует два типа пакетов: короткий и длинный. Короткий пакет в частности сигнализирует о начале и конце кадра. В длинном пакете происходит передача данных, обычно одна строка изображения.

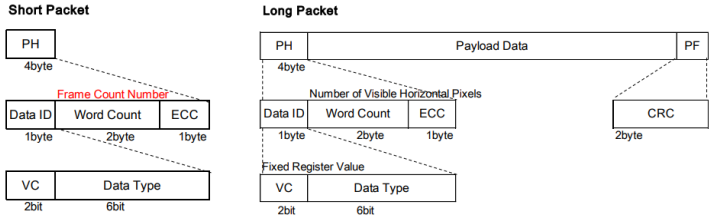


Рисунок 7 – Структура кадров CSI-2

Оба типа пакета начинаются с заголовка, в который входит:

- Синхрослово 0xB8
- 1 байт информации о данных: 2 бита номера виртуального канала (0-3, т.е. 4 независимых канала передачи), 6 бит типа данных (0x00 – начало кадра, 0x01 – конец кадра, 0x24 – RGB888 и т.д).
- 2 байта с числом количества передаваемых данных (в коротком кадре опционально можно передавать номер кадра)
- 1 байт с кодом коррекции ошибок

Данные передаются с наименее значащего бита, поэтому каждая HS передача начинается с последовательности 0001 1101 (0xB8).

3. Описание IP блоков

IDE Gowin предоставляет доступ к IP блокам, необходимым для постоения MIPI передатчика.

Было использовано 3 ядра:

- MIPI Pixel-to-Byte Converter
- MIPI DSI/CSI-2 Transmitter
- MIPI TX Advance

MIPI Pixel-to-Byte Converter используется для преобразования стандартных сигналов передачи видео (hs, vs, de, data, clk) в сигналы и данные для формирования кадра CSI-2.

I_FV – Frame Valid (высокий уровень в течении всей передачи кадра изображения)

I_LV – Line valid (высокий уровень в течении всей передачи строки)

I_PIXEL – данные

I_PIXEL_CLK – тактовая частота данных пикселя

I_BYTE_CLK – выходная частота для передачи данных дальше

Соотношение тактовых частот рассчитывается по формуле

$$\frac{Byte\ Clock}{Pixel\ Clock} = \frac{bits\ per\ pixel \times pixel\ per\ pixclk}{number\ of\ TX\ lanes \times TX\ gear}$$

Для типа данных RGB888, 1 пиксель на такт пиксель-клока, режима передачи 8:1 по двум линиям передачи соотношение составит:

$$\frac{Byte\ Clock}{Pixel\ Clock} = \frac{24 \times 1}{2 \times 8} = \frac{3}{2}$$

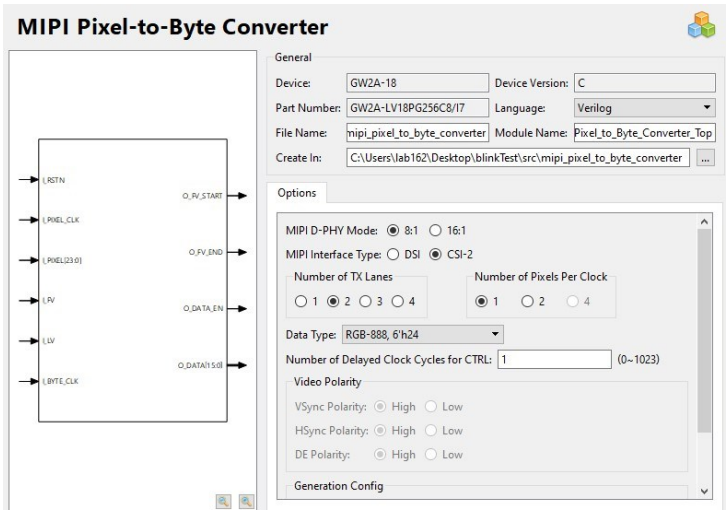


Рисунок 8 – IP- ядро конвертора пикселей в байты

Полученные сигналы поступают в ядро для формирования кадров MIPI CSI-2 Transmitter. При создании ядра нужно указать текущую частоту тактирования байт данных и установить необходимые тайминги. При слишком низких значениях таймингов среда разработки будет выдавать ошибку.

Большая часть входных сигналов была получена из предыдущего ядра, необходимо дополнительно указать:

- I_WC – Word Count – количество байт в строке кадра (3*1024 = 3840)
- I_VC – Virtual Channel – виртуальный канал (0x00)
- I_DT – Data Type – тип данных (для RGB888 0x24)

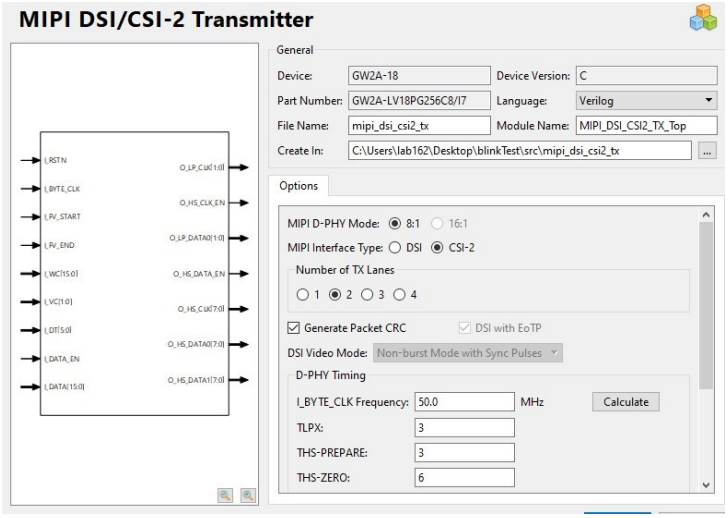


Рисунок 9 – Передатчик-формирователь сигналов управления интерфейсом

IP ядро формирует данные для дальнейшей передачи, включая заголовок и контрольную сумму, сигналы управления.

Далее данные поступают в ядро MIPI TX Advance. В среде разработки там же присутствует устаревшее ядро с названием MIPI TX. Большая часть входов подключается к выходам предыдущего блока. На входы lp_*_dir можно подать единицы, sclk – тактовая байт данных (byte clock), CLKOP и CLKOS – учетверенная частота байт данных, с фазами 0° и 90°. Выходные сигналы ядра подключаются к соответствующим выводам ПЛИС.

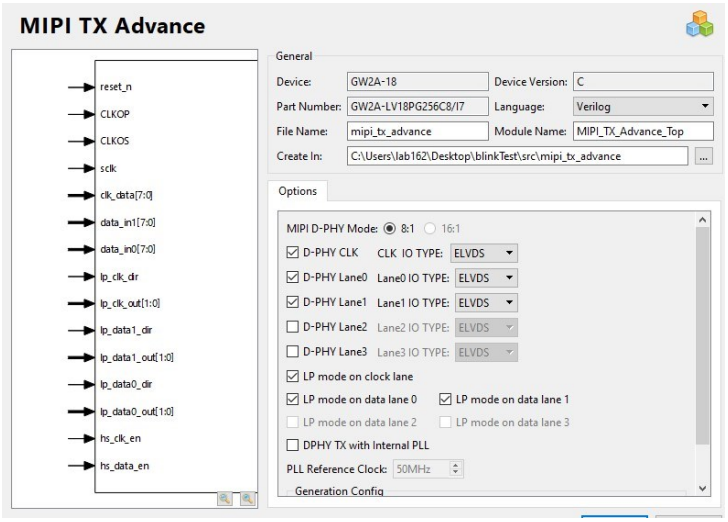


Рисунок 10 – Ядро передачи MIPI D-PHY

4. Подключение к Raspberry Pi

Сложность подключения камеры в Raspberry Pi заключается в закрытости исходного кода видеоядра и наличии крипто-чипов на оригинальных камерах Raspberry.

Поэтому для получения данных использовалась утилита RaspiRaw с добавленным плагином GStreamer, позволяющий строить пайплайны для дальнейшей работы с видео.

Так же в RaspiRaw была переписана конфигурация камеры imx219 для работы с разработанной платой. Тип данных был изменен на GRB888, размер изображения 1280x720, и включено ручное управление согласующими резисторами на линии тактов.

Итоги

В ходе работы над проектом были определены плюсы и минусы использованного модуля.

Плюсы:

- наличие доступных IP-ядер

- дешевизна

- полная документация к самому модулю, включая длины дорожек

Минусы:

- отсутствие документации к части IP ядер

- поддержка игнорировала просьбы выслать лицензию для Standart версии, пока не указал другую страну и не отправил запрос под VPN (при желании этот шаг возможно пропустить, воспользовавшись кряком, но всё же)

- менее проработанная IDE по сравнению с конкурентами

В целом работа с данными ПЛИС оставила положительные впечатления. Полный код передатчика можно найти на GitHub по ссылке https://github.com/AleksandrYMin/GoWinStereoCam

Реализация Avalon-MM Master в виде конечного автомата на VHDL

Бортников Анатолий Юрьевич
генеральный директор компании ООО "РСВ ЭЛЕКТРОНИКС", кандидат физ.- мат. наук.

Телеграм: @PCBproj

Обсуждение и комментарии :: ссылка

Введение

Шина Avalon-MM является одной из стандартных шин передачи данных, используемых в ПЛИС фирмы Intel. Использование этой шины в своих модулях для передачи данных существенно повышает их возможность повторного применения и повышает надежность проектов. Также упрощается интеграция модулей в проект с помощью Platform Designer.

Принципы работы шины Avalon-MM.

Шина Avalon-MM используется для передачи данных между модулями проекта. Она позволяет выполнять запись данных в модуль либо чтение данных из модуля обращаясь к нему как к блоку памяти.

На шине Avalon-MM существует определенная иерархия модулей. Обязательно должен быть контроллер - Master, который управляет всеми транзакциями передачи данных по шине. Остальные модули выступают в роли подчиненных - Slave. Каждый Slave на шине имеет определенный адрес или диапазон адресов, в которые Master может писать данные или читать из этих адресов. На рис. 1 представлен вариант конфигурации шины Avalon-MM.

В качестве мастера часто выступает процессор NIOS II либо может использоваться другой модуль в качестве процессора. Если использовать NIOS II то проблем с подключением к шине Avalon-MM не возникает. Так как он уже имеет встроенный интерфейс Avalon-MM Master. Если же есть необходимость использовать процессор без интерфейса Avalon-MM (например когда процессор пишется пользователем под конкретную задачу) то для его подключения к шине Avalon-MM требуется реализация интерфейса Avalon-MM Master.

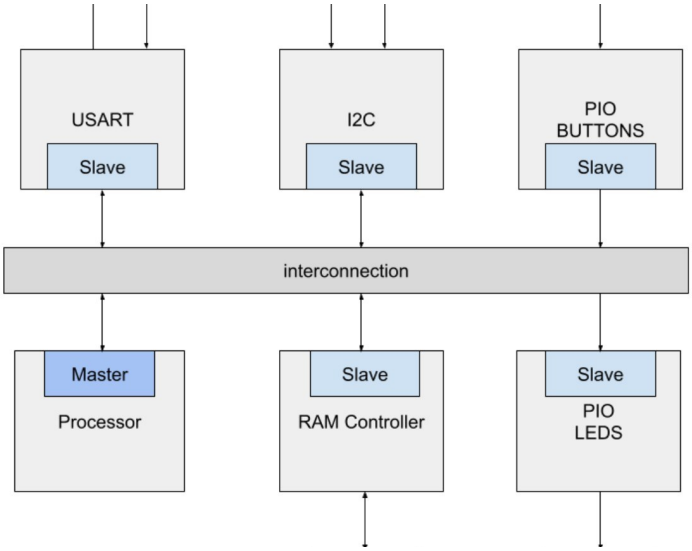


Рис 1. Вариант конфигурации шины Avalon-MM

Так как для управления периферией предполагается выполнение операций записи и чтения данных, рассмотрим минимально необходимый для этого набор сигналов интерфейса Avalon-MM Master, представленный в таблице 1.

Как было упомянуто выше, все передачи данных инициализирует Master. Передачи данных осуществляется словами. Разрядность слов настраивается дискретными значениями в диапазоне от 8 бит до 1024 бит. Активные байты в передаваемых словах обозначаются маской в сигналах byteenable или byteenable_n.

Slave может приостановить передачу данных выставив в единицу сигнал waitrequest. Тогда, пока сигнал waitrequest равен 1, Master находится в режиме ожидания и не имеет права переключать сигналы интерфейса. Транзакция может завершиться только когда Slave сбросит сигнал waitrequest в 0 на рис. 2 представлены временные диаграммы записи и чтения данных.

Таблица 1. Сигналы шины Avalon-MM, минимальный набор

Название	разряды	направление	описание
address	1-32	выход	address представляет адрес байта независимо от ширины шины данных. Значение адреса должно быть выровнено по ширине шины данных. Для записи отдельных байтов с шины данных требуется использовать сигналы byteenable. Мастер всегда выставляет адрес независимо от ширины шины данных. Система межсоединений конвертирует этот адрес в адрес слов в адресном пространстве Slave-модуля.
read, read_n	1	выход	Сигнал запроса операции чтения. Не требуется, если Master никогда не инициализирует операцию чтения. Но если есть сигналы read/read_n то readdata обязательны.
readdata	8, 16, 32, 64, 128,	вход	Сигналы для чтения данных.
write write_n	1	выход	Сигнал запроса операции записи. Не требуется, если Master никогда не инициализирует операцию записи данных. Но если есть write/write_n то сигналы writedata обязательны.
writedata	8, 16, 32, 64, 128, 256, 512,	вход	Сигналы для передачи данных для записи в Slave. Если присутствуют writedata и readdata, то они должны быть одинаковой разрядности.
byteenable byteenable_n	1, 2, 4, 8, 16, 32, 64, 128	выход	Сигналы активизируют отдельные байты данных при передаче, если разрядность шины данных больше 8. Каждый бит в byteenable соответствует отдельному байту в шине данных как при чтении так и при записи. Единица в бите byteenable показывает, будет ли передан этот байт по шине данных. Остальные байты в шине данных должны игнорироваться Slave-ом. Если требуется передать больше чем 1 байт по шине данных, то все байты должны быть выставлены. Количество передаваемых байт должно быть равно степени 2. Доступны следующие варианты значения сигналов byteenable и количество передаваемых байт: 1111 - передача 4-х байтов (32 бита); 0011 - передача младших двух байт (16 бит); 1100 - передача старших двух байт (16 бит); 0001 - передача младшего (первого) байта; 0010 - передача второго байта; 0100 - передача третье байта; 1000 - передача четвертого байта;
waitrequest waitrequest_n	1	вход	Сигнал ожидания завершения транзакции. waitrequest выставляется Svale-ом, в случае, если он не может в данный момент принять входные данные или выдать данные для чтения. Master ожидает пока waitrequest не сбросится, чтобы завершить транзакцию. В это время Master не должен переключать сигналы на шине Avalon-MM.

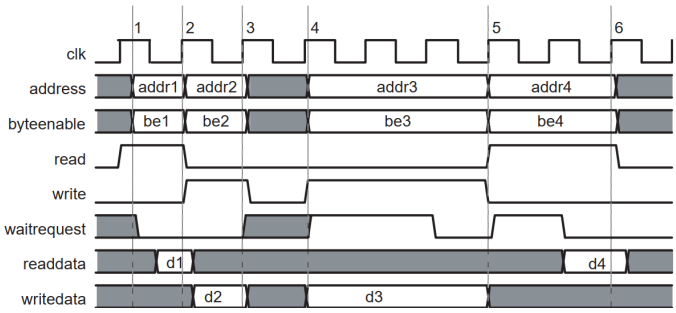


Рис. 2 Временные диаграммы записи и чтения данных по шине Avalon-MM.

1. Master начинает чтение данных выставляя сигналы address, byteenable и read. Slave выдает данные на readdata в течение следующего такта clk.
2. Master считывает данные с readdata и сбрасывает сигнал read, заканчивая тем самым операцию чтения данных. И тут же начинает операцию записи данных выставляя сигналы address, byteenable, write и данные на writedata.
3. Сигнал waitrequest не выставляется в единицу по следующему фронту clk, следовательно по этому фронту заканчивается операция записи. Master сбрасывает сигнал write.
4. Master выставляет address, byteenable, writedata и write начиная следующую операцию записи. Тут же Slave поднимает сигнал waitrequest, сообщая, что он не готов в данный момент обрабатывать данные. Поэтому Master удерживает без изменения все сигналы шины Avalon-MM.
5. waitrequest сброшен, и тут операция записи завершается. И сразу Master начинает следующую операцию чтения выставляя сигналы read, address, byteenable. Slave, в свою очередь, поднимает сигнал waitrequest, сообщая, что он занят выполнением другой операции и не может в данный момент выставить данные на выход readdata. На следующем такте Slave выставляет данные на шину readdata и сбрасывает сигнал waitrequest.
6. waitreqiuest сброшен на следующем фронте сигнала clk, следовательно на этом фронте clk операция чтения завершается.

Реализация интерфейса Avalon-MM Master в виде конечного автомата

Если посмотреть внимательно на временные диаграммы записи чтения данных изображенных на рис. 2, то можно увидеть, что интерфейс при каждой транзакции проходит одни и те же состояния. Следовательно его можно реализовать в виде конечного автомата. Но сначала определим порты для нашего интерфейса Avalon-MM Master.

Порты интерфейса Avalon-MM Master

Представим наш интерфейс Avalon-MM Master как отдельный модуль, встраиваемый в какой-либо процессор или контроллер в следующем виде рис. 3.

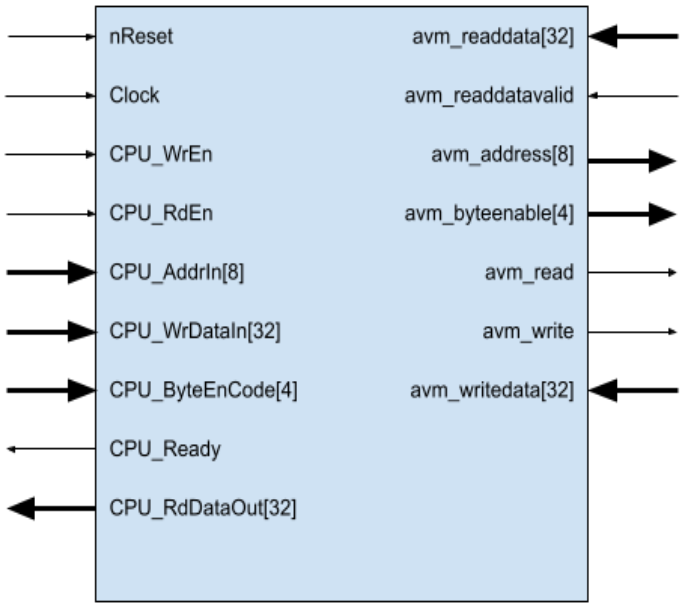


Рис. 3 Графическое обозначение модуля Avalon-MM Master.

Описание портов ввода-вывода на языке VHDL для компонента Avalon-MM Master, графическое представление которого изображено на рис. 3, приведено ниже в листинге 1.

Листинг 1: Описание портов ввода\вывода для Avalon-MM Master

```
PORT(
  nReset          : IN  STD_LOGIC;
  Clock           : IN  STD_LOGIC;

  ------------ Processor Interface --------------------
  CPU_WrEn        : IN  STD_LOGIC;
  CPU_RdEn        : IN  STD_LOGIC;
  CPU_AddrIn      : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
  CPU_WrDataIn    : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
  CPU_ByteEnCode  : IN  STD_LOGIC_VECTOR( 3 DOWNTO 0 );
  CPU_Ready       : OUT STD_LOGIC;
  CPU_RdDataOut   : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );

  ---------- Avalon-MM Master interface --------------
  avm_readdata      : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
  avm_readdatavalid : IN  STD_LOGIC;
  avm_address       : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 );
  avm_byteenable    : OUT STD_LOGIC_VECTOR( 3 DOWNTO 0 );
  avm_read          : OUT STD_LOGIC;
  avm_write         : OUT STD_LOGIC;
  avm_writedata     : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 )
);
```

Согласно таблице 1, разрядность и шин данных avm_writedata и avm_readdata равны между собой. Разрядность сигналов avm_byteenable равна разрядность шин данных деленной на 8. Каждый бит в avm_byteenable ставится в соответствии байту на шине данных.

Разрядность шина адреса avm_address составляет в данном случае 8 бит. Разрядность шины адреса выбирается в зависимости от того, насколько большие области адресов памяти выделяются для компонентов периферии.

Если в проекте не используются модули встроенной памяти или контроллеры доступа к внешней памяти SDRAM, DDR-DDR3, то скорее всего большая разрядность шины адреса не потребуется.

Однако, если предполагается работа с перечисленными выше контроллера доступа к памяти, то может потребоваться достаточно большая разрядность шины адреса: 8, 16, 24, 32 бита.

Это связано с тем, что для модулей периферии выделяются относительно не большие объемы памяти - буквально несколько регистров. Ниже перечислены самые распространенные регистры, которые присутствуют практически в каждом периферийном модуле:

- регистр записи данных;
- регистр чтения данных
- регистр управления и настройки
- регистр статуса

Для модулей работы с памятью, в свою очередь, выделяются гораздо большие массивы памяти, что требует большей разрядности шины адреса.

Описание конечного автомата

На рис. 4 представлена диаграмма состояний конечного автомата интерфейса Avalon-MM Master.

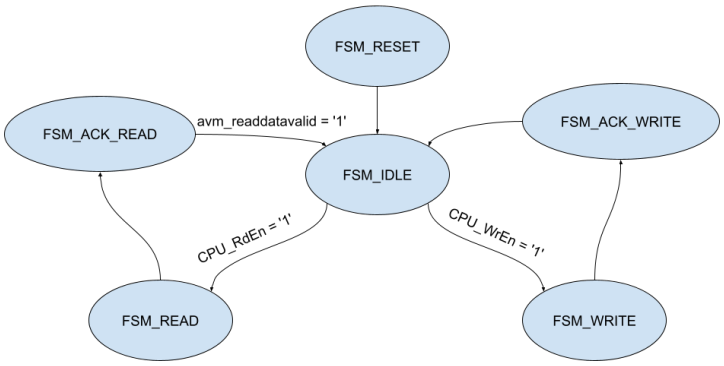


Рис. 4 Диаграмма состояний
конечного автомата интерфейса Avalon-MM Master

После подачи питания конечный автомат попадает в состояние сброса FSM_RESET. Где сбрасываются в исходные состояния все внутренние переменные и сигналы шины Avalon-MM.

На следующем такте конечный автомат переходит в состояние ожидания FSM_IDLE. Это исходное состояние автомата для работы с шиной Avalon-MM. В этом состоянии автомат ожидает команду от процессора. В листинге 2 представлен исходный код, описывающий это состояние на VHDL.

Листинг 2. Исходный код реализации состояния FSM_IDLE

```
WHEN FSM_IDLE =>
  IF( CPU_WrEn = '1' ) THEN
   MasterAddr  <= CPU_AddrIn;
   MasterData  <= CPU_WrDataIn;
   FSM_MasterState <= FSM_WRITE;
  ELSIF( CPU_RdEn = '1' ) THEN
   MasterAddr   <= CPU_AddrIn;
   FSM_MasterState <= FSM_READ;
  ELSE
   MasterAddr      <= ( OTHERS => '0' );
   MasterData      <= ( OTHERS => '0' );
   avm_write          <= '0';
   avm_read           <= '0';
   avm_byteenable     <= ( OTHERS => '0' );
   ReadyTmp           <= '0';
  END IF;
```

По сигналам CPU_WrEn и CPU_RdEn происходит предварительное защелкивание адреса и данных с процессора во внутренние переменные конечного автомата.

При появлении единицы на линии CPU_RdEn, конечный автомат переходит в состояние FSM_READ - состояние чтения данных по шине Avalon-MM. Процессор должен вместе с единицей на линии CPU_RdEn выставить адрес на входные линии CPU_AddrIn, с которого он запрашивает чтение данных. И в это же время процессор должен задать код на шине CPU_ByteEnCode, чтобы определить, какие байты на шине данных будут активны.

В состоянии FSM_READ конечный автомат дублирует адрес на шину avm_address, повторяет код с CPU_ByteEnCode на линии avm_byteenable и выставляет единицу на линию avm_read, в соответствии с временными диаграммами на рис. 2. В листинге 3 показан исходный код этого состояния на VHDL

Листинг 3. Исходный код реализации состояния FSM_READ

```
WHEN FSM_READ =>
  avm_write          <= '0';
  avm_read           <= '1';
  avm_address        <= MasterAddr;
  avm_byteenable     <= CPU_ByteEnCode;
  FSM_MasterState     <= FSM_ACK_READ;
```

На следующем такте конечный автомат переходит в состояние FSM_ACK_READ - подтверждение чтения данных. В этом состоянии автомат проверяет значение на линии avm_readdatavalid.

Если значение avm_readdatavalid равно единице, значит на линии avm_readdata доступны верные данные. В таком случае эти данные дублируются на выход CPU_RdDataOut и на линию CPU_Ready выставляется единица, как сигнал для процессора, что с интерфейса Avalon-MM Master доступны новые данные для чтения. На следующем такте автомат переходит в исходное состояние FSM_IDLE.

Если же значение avm_readdatavalid равно нулю, то на линию CPU_Ready выставляется ноль и на следующем такте автомат переходит также в исходное состояние FSM_IDLE.

В листинге 4 представлен исходный код реализации состояния FSM_ACK_READ на VHDL.

Листинг 4. Исходный код реализации состояния FSM_ACK_READ

```
WHEN FSM_ACK_READ =>
  IF( avm_readdatavalid = '1' ) THEN
   ReadyTmp     <= '1';
   RdDataOutTmp <= avm_readdata;
   FSM_MasterState <= FSM_IDLE;
  ELSE
   FSM_MasterState <= FSM_IDLE;
   ReadyTmp     <= '0';
   RdDataOutTmp <= ( OTHERS => '0' );
  END IF;
```

Из состояние FSM_IDLE конечный автомат по единице на входной линии CPU_WrEn переходит в состояние FSM_WRITE. В этом состоянии дублируется значение адреса с линии CPU_AddrIn и код с линии CPU_ByteEnCode на выходные линии avm_address и avm_byteenable соответственно. И выставляется единица

на линию avm_write. Исходный код реализации этого состояния представлен в листинге 5.

Листинг 5: Исходный код реализации состояния FSM_WRITE.

```
WHEN FSM_WRITE =>
  avm_write          <= '1';
  avm_read           <= '0';
  avm_address        <= MasterAddr;
  avm_byteenable     <= CPU_ByteEnCode;
  avm_writedata      <= MasterData;
  FSM_MasterState    <= FSM_ACK_WRITE;
```

На следующем такте автомат переходит в состояние FSM_ACK_WRITE, в котором сбрасывается в 0 сигнал на линии avm_write. Это приводит к завершению операции записи данных в периферийный модуль. Исходный код реализации состояния FSM_ACK_WRITE представлен в листинге 6.

Листинг 6. Исходный код реализации состояния FSM_ACK_WRITE.

```
WHEN FSM_ACK_WRITE =>
  avm_write <= '0';
  FSM_MasterState <= FSM_IDLE;
```

На следующем такте автомат переходит в исходное состояние FSM_IDLE для ожидания новой команды от процессора.

Обработка waitrequest

Конечно не всегда периферийный модуль может за один такт успеть защелкнуть записываемые в него данные либо успеть выдать на следующем такте запрашиваемые в него данные. На этот случай шина Avalon-MM имеет сигнал waitrequest.

Сигнал waitrequest управляется интерфейсом Avalon-MM Slave на периферийном модуле (см. рис. 2). Если периферийный модуль не может в данный момент отреагировать на запрос от Avalon-MM Master, то он выставляет 1 на линию waitrequest. Это приводит к тому, что Avalon-MM Master должен удерживать все сигналы на линии Avalon-MM в текущем состоянии, пока периферийный модуль не сбросить waitrequest в 0. И только тогда Avalon-MM Master сможет завершить текущую операцию записи или чтения.

Обработку сигнала waitrequest лучше всего добавить в состояния FSM_ACK_READ и FSM_ACK_WRITE. Так как в этих состояниях автомат уже выставил все сигналы на шине Avalon-MM, то тут нам только потребуется проверять значения сигнала waitrequest. Пока он равен 1, автомат должен находиться в текущем состоянии, а как только waitrequest будет сброшен в 0, автомат сможет завершить текущую операцию и перейти в состояние FSM_IDLE. Исходные коды реализации состояний FSM_ACK_READ и FSM_ACK_WRITE с обработкой сигнала waitrequest представлены в листинге 7 и 8 соответственно.

Листинг 7. Исходный код реализации сосотяния FSM_ACK_READ с обработкой сигнала waitrequest.

```
WHEN FSM_ACK_READ =>
  IF( avm_readdatavalid = '1' ) THEN
   ReadyTmp       <= '1';
   RdDataOutTmp   <= avm_readdata;
   avm_read         <= '0';
   FSM_MasterState <= FSM_IDLE;
  ELSE
   ReadyTmp      <= '0';
   RdDataOutTmp <= ( OTHERS => '0' );
   IF( avm_waitrequest = '0' ) THEN
    IF (wait_cycle > 0) THEN
     wait_cycle <= wait_cycle - 1;
    ELSE
     FSM_MasterState <= FSM_IDLE;
    END IF;
   ELSE
    IF( AckCounter > 0 )THEN
     AckCounter := ( AckCounter - 1 );
    ELSE
     FSM_MasterState <= FSM_IDLE;
    END IF;
   END IF;
  END IF;
```

Листинг 8. Исходный код реализации сосотяния FSM_ACK_WRITE с обработкой сигнала waitrequest.

```
WHEN FSM_ACK_WRITE =>
  IF( avm_waitrequest = '0' ) THEN
   avm_write <= '0';
   FSM_MasterState <= FSM_IDLE;
  ELSE
   IF( AckCounter > 0 )THEN
    AckCounter := ( AckCounter - 1 );
   ELSE
    avm_write <= '0';
    FSM_MasterState <= FSM_IDLE;
   END IF;
  END IF;
```

Нужно отметить, что в условие проверки сигнала waitrequest полезно добавить счетчик-таймер, по истечению которого, если периферийный модуль так и не сбросит сигнал waitrequest в 0, то автомат вернется в исходное состояние FSM_IDLE. Таким образом возможное зависание периферийного модуля не приведет к зависанию интерфейса Avalon-MM Master.

В нашем случае мы добавили счетчик AckCounter, который считает от 15 до 0. В состоянии FSM_IDLE значение этого счетчика выставляется равное 15.

Так же нужно отметить, что в состояние FSM_ACK_READ была добавилена задержка в 1 такт на счетчике wait_cycle. Это было сделано для того, чтобы автомат

задержался в состоянии FSM_ACK_READ на 1 такт дольше, чтобы дождаться появления 1 на линии avm_readdatavalid.

Отладка конечного автомата Avalon-MM Master

Для отладки работы конечного автомата был написан testbench, достаточно простой. Все окружение, кроме тестируемого интерфейса Avalon-MM Master состояло из конечного автомата, который эмулировал команды процессора - FSM_CPU и модуля встроенной памяти On-Chip Memory с интерфейсом Avalon-MM Slave.

Автомат процессора последовательно задавал команды на запись и чтение данных. Встроенная память была изначально проинициализирована тестовыми данными.

На рис. 5 представлены временные диаграммы работы компонента Avalon-MM Master с встроенной памятью через шину Avalon-MM.

Когда процессор начинает операцию записи, он выставляет на линии TB_CPU_WrEn, TB_CPU_AddrIN, TB_CPU_WrDataIn соответсвенно сигнал записи, адрес ячейки (0xAD) и данные, которые требуется записать в память (0xDADA0505).

В свою очередь автомат интерфейса Avalon-MM Master переходит в состояние FSM_WRITE и выставляет на шину Avalon-MM сигнал TB_avm_write равный 1 и дублирует адрес и данные на линии TB_avm_address и TB_avm_writedata соответвтенно.

На следующем такте On-Chip Memory выдает 1 на линию TB_waitrequest, сигнализируя о процессе записи данных в память. После чего процедура записи заканчивается. Автомат интерфейса Avalon-MM Master переходит в состояние FSM_IDLE.

Для начала операции чтения процессор выставляет на линии TB_CPU_RdEn интерфейса Avalon-MM Master единицу и на линии TB_CPU_AddrIN адрес ячейки (0xBB) для чтения данных.

Автомат интерфейса Avalon-MM Master переходит в состояние FSM_READ и выставляет единицу на линии TB_avm_read а так же дублирует адрес ячейки на шину адреса TB_avm_address и выставляет код активных байтов (0xF) на линии TB_avm_byteenable.

На следующем такте On-Chip Memory выдает 1 на линию TB_waitrequest, сигнализируя о процессе чтения данных из памяти. И в следующий такт считаные данные выставыляются на линию TB_avm_readdata (0x1A2B3C4D) и одновременно с этим выставляется единица на линию TB_avm_readdatavalid, сигнализирующая, что выставлены валидные данные. После чего автомат интерфеса Avalon-MM Master выставляет единицу на линию TB_CPU_Ready, показывая что доступны новые данные для процессора и дублирует данные с линии TB_avm_readdata на выход TB_CPU_RdDataOut. После чего операция чтения завершается.

Заключение

В данной статье был рассмотрен вариант реализации конечного автомата для интерфейса Avalon-MM Master. Данная реализация позволяет встраивать этот интерфейс в свои компоненты, что приводит к повышению степени их повторного использования и упрощает проектирование систем на ПЛИС с использованием инструмента Platform Designer.

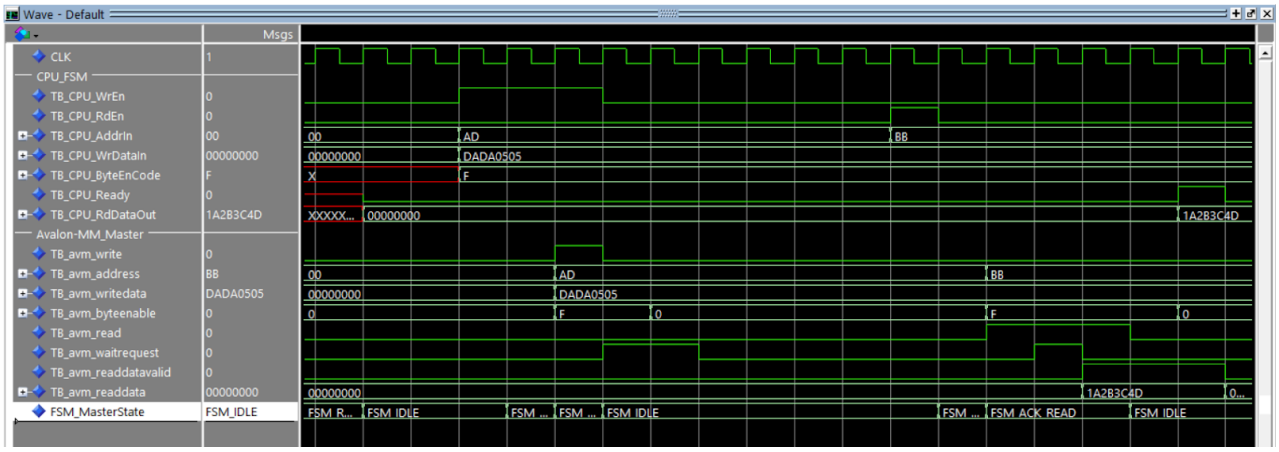


Рис. 5. Временные диаграммы работы компонента Avalon-MM Master с встроенной памятью через шину Avalon-MM (приставка TB_ обозначает сигналы из testbench)

PCBteach

ООО "РСВ Электроникс"

г. Санкт-Петербург

Обучаем технический персонал и IT специалистов. Лицензия на образовательную деятельность в сфере электроники.

Наши образовательные программы:

Основы проектирования устройств на базе ПЛИС

Основы цифровой схемотехники. Основы работы в среде разработки Intel Quartus Prime. Основы языков описания цифровых схем Verilog и VHDL. Моделирование и отладка проектов на ПЛИС.

Программирование микроконтроллеров

Архитектура микроконтроллеров, периферийные модули ARM STM32. Основы работы в среде разработки Segger Embedded Studio for ARM. Написание программ для STM32 с применением CMSIS, а также FreeRTOS и отладка проекта с помощью внутрисхемного отладчика.

Практические навыки работы инженера-радиоэлектронщика

Базовые знания и навыки по разработке электрических принципиальных схем устройств и трассировке печатных плат. В качестве инструмента используется САПР KiCAD. Полученные на курсе знания остаются актуальны при использовании любой САПР.

Практические навыки трассировки печатных плат

Систематизированные знания и навыки по трассировке печатных плат любой сложности, методики трассировки многослойных печатных плат, трассировки высокоскоростных параллельных интерфейсов и последовательных высокоскоростных линий передачи данных.

Программирование микроконтроллеров с использованием ОСРВ FreeRTOS

Основы программирования микроконтроллеров с использование ОСРВ FreeRTOS: способы управления задачами, режимы распределения памяти, методы управления очередью, использование программных таймеров, управление прерываниями, способы распределения ресурсов.

Оптимизация и отладка проектов ПЛИС внутри микросхемы

Программа курса включает в себя навыки оптимизации временных характеристик проекта ПЛИС с использованием Quartus TimeQuest и TCL-скриптов. В курсе также рассматриваются методики отладки проекта внутри микросхемы ПЛИС с использованием Quartus SignalTap.

Наши клиенты:

РПКБ, ТМХ ИНЖИНИРИНГ, Модуль, АТОМТЕХ, Form, ТУСУР TUSUR UNIVERSITY, НПП ПРИМА, Центр ВОСПИ, МИРТЕК изобретая будущее

Наши преимущества:

- ✓ Дистанционный формат обучения без отрыва от работы.
- ✓ В процессе обучения обучающийся разработает электронное устройство и напишет ПО для МК или ПЛИС.
- ✓ Получение документа о профессиональной переподготовке или о повышении квалификации установленного образца.
- ✓ Возможность оплаты обучения в рассрочку без %.

После окончания курсов Вы будете уметь:

- Разрабатывать электрические принципиальные схемы и разводить печатные платы.
- Понимать документацию на электронные компоненты и заказывать печатные платы на производстве.
- Монтировать, собирать электронные устройства и управлять ими.
- Писать базовые программы для микроконтроллеров и разбираться в архитектуре микроконтроллеров.
- Пользоваться программой Debugger для отладки работы программы, работать в универсальной среде разработки для STM32 Segger Embedded Studio.
- Работать с основными периферийными модулями, с операционной системой реального времени – FreeRTOS.
- Пользоваться инструментами среды разработки Intel Quartus Prime.
- Описывать внутреннюю архитектуру ПЛИС на языках Verilog и VHDL
- Описывать внутреннюю архитектуру ПЛИС с использованием IP-ядер в конструкторе систем на кристалле QSYS.
- Пользоваться инструментами моделирования и отладки ModelSIM.

Контактная информация:

8-800-301-66-34
www.pcbteach.ru
info@pcbteach.ru
https://vk.com/pcbteach
https://t.me/stm32_i_plis

Интеграция гигабитного последовательного интерфейса на основе стандарта JESD204B: расширение горизонтов передачи данных в ПЛИС

Кашпурович Владимир Владимирович

Telegram: @AshhCatt
e-mail: vvkashpurovich@gmail.com

Обсуждение и комментарии :: ссылка

Аннотация

Современные требования к проектированию коммерческих продуктов усиливают необходимость оптимизации этапов разработки и сокращения времени прототипирования. При этом для некоторых приложений важно обеспечить легкую масштабируемость разработки. В контексте конвертации аналогового сигнала в цифровой одним из наиболее эффективных методов оптимизации как трассирования платы, так и использования предоставленного ресурса контактов является замена параллельного интерфейса передачи данных последовательным. Одним из способов реализации подобного решения на практике является применение гигабитного последовательного интерфейса на основе стандарта JESD204B. Сборка данного интерфейса на основе IP-ядер из открытого репозитория позволяет ускорить разработку и изготовление платы, обеспечив легкую масштабируемость всей системы регистрации или генерации данных. В данной работе приведено описание рабочего дизайна гигабитного последовательного интерфейса на основе стандарта JESD204B, собранного на IP-ядрах от Analog Devices. По результатам экспериментов была достигнута скорость передачи данных в 5 Гбит/с при частоте оцифровки 250 МГц и разрядности АЦП 14 бит. Однако нужно учитывать, что использование подобного интерфейса на практике сопряжено с увеличением задержки в регистрации данных, а также с использованием части ресурсов ПЛИС, включая скоростные трансиверы.

Ключевые слова: скоростная передача данных, гигабитный последовательный интерфейс, стандарт JESD204, ПЛИС.

1. Введение

В современном мире все большее распространение и развитие получают компактные и дешевые технологии измерения высокой точности, что в первую очередь сказывается на скорости и объеме генерации новых данных. Эффект от такого развития можно наблюдать, например, в требованиях, предъявляемых к современным измерительным системам, а также к системам обработки и анализа данных, используемым в науке, медицине, промышленности и цифровой коммуникации. Среди них стоит выделить необходимость удешевления себестоимости при сохранении или даже улучшении характеристик этих систем, таких как: уменьшение габаритных размеров приборов, серверов, отдельных элементов печатных плат; уменьшение уровня энергопотребления и тепловыделения; увеличение скорости обработки или передачи данных.

Важно отметить, что характеристики электронного оборудования формируются не только на основе используемой элементной базы. Необходимо также продумать организацию правильного взаимодействия между этими элементами. Поэтому для соответствия некоторым из предъявляемых к оборудованию требованиям необходима оптимизация передачи данных как внутри него, так и за его пределами. При этом с коммерческой точки зрения важно обеспечить простоту и дешевизну масштабируемости такого оборудования для быстрого его распространения и развертывания. В контексте схемотехники одним из самых очевидных способов достижения указанных аспектов является упрощение трассировки печатных плат, а также оптимизация использования контактов процессоров или контроллеров.

Самым явным способом достижения поставленной цели является замена параллельных интерфейсов передачи данных между конвертерами и контроллерами на последовательные, при сохранении конечной скорости передачи данных. Например, замена параллельного интерфейса передачи данных от 14-битного АЦП [1] на последовательный позволит освободить множество контактов для других задач. Но только в случае увеличения скорости передачи по оставшейся линии в 14 раз.

Одним из стандартов, предназначенных для внедрения высокоскоростного последовательного интерфейса, является JESD204 [2]. Данный стандарт, разработанный

и используемый для организации гигабитного последовательного обмена данных, озаглавлен как «Serial Interface for Data Converters». Он является открытым, развивается и поддерживается ассоциацией JEDEC (Joint Electron Device Engineering Council). На сегодняшний день представлено 4 версии стандарта JESD204. Ниже (таблица 1) приведены некоторые из их основных особенностей, к которым апеллируют при рассмотрении данного стандарта.

Из таблицы видно, что стандарт JESD204 с каждой ревизией только набирает функционал, тем самым расширяя свою практическую значимость и применимость. А различные аспекты эффективности и компактности последовательного интерфейса на его основе позитивно сказываются на масштабируемости разрабатываемых систем. Это побуждает к рассмотрению интерфейсов на основе указанного стандарта в качестве возможного решения при конвертации аналоговых сигналов в цифровые и обратно. К недостаткам этого решения нужно отнести в первую очередь сложность самой системы организации передачи данных согласно данному стандарту и относительно большую задержку регистрации данных, формирующуюся из-за необходимости в сериализации и десериализации данных на этапе передачи.

Таблица 1. Версии стандарта JESD204.

Характеристика	JESD204	JESD204A	JESD204B	JESD204C
Год публикации	2006	2008	2011	2017
Скорость передачи по линии	до 3.125 Гбит/с	до 3.125 Гбит/с	до 12.5 Гбит/с	До 32.45 Гбит/с
Поддержка нескольких линий	-	+	+	+
Синхронизация между линиями	-	+	+	+
Синхронизация устройств	-	+	+	+
Контроль задержки	-	-	+	+

Данная работа была посвящена описанию процесса апробации гигабитной последовательной передачи данных по интерфейсу на основе стандарта JESD204B без использования детерминированной задержки между АЦП AD9250 (Analog Devices) и системой на кристалле (СнК) семейства Zynq-7000 (Xilinx).

2. Описание экспериментальной схемы

Разработанный проект системы оцифровки и передачи данных по стандарту JESD204B (рисунок 1) включает в себя АЦП, осциллятор для АЦП и ПЛИС. Для управления схемой и взаимодействия с внешними системами предполагается использование процессора. Для реализации дополнительных возможностей гибкой конфигурации всей системы под различные задачи в схему проекта был также добавлен блок DDR памяти. Непосредственная обработка данных АЦП производится в соответствующем блоке на стороне ПЛИС. После чего осуществляется их передача либо процессору, либо в DDR память. Наилучшим доступным инструментом для решения описанной задачи являлась СнК семейства Zynq-7000.

На момент изготовления рабочего стенда на основе описанной схемы доступная элементная база включала в себя плату Avnet PicoZed 7Z030 SOM с кристаллом XC7Z030-1SBG485C и объемом DDR3 памяти в 1 ГБ. Упомянутый кристалл является хорошим решением поставленной задачи поскольку содержит 4 трансивера [3] с поддержкой скоростей передачи по линии до 6.6 Гбит/с. Альтернативные СнК либо были недоступны, либо содержат значительно большее количество трансиверов (а значит и стоят дороже), либо не поддерживают необходимую скорость передачи по линии.

Рисунок 1. Схема организации контроля и передачи оцифрованных данных.

Для генерации опорного тактового сигнала использовался программируемый по интерфейсу IIC тактовый генератор Si-5332.

Устройство от Analog Devices AD9250 представляет собой сдвоенный конфигурируемый по интерфейсу SPI аналого-цифровой преобразователь с предельной частотой оцифровки 250 МГц и разрешением 14 бит. В документации на конвертер представлена подробная схема цифровой обработки зарегистрированного преобразователем сигнала согласно стандарту JESD204B и широкие возможности его конфигурации. Преобразователь был настроен так, чтобы на выходе скорость передачи данных по линии была максимальной и составляла 5 Гбит/с. В данной работе не рассматривался функционал, связанный с использованием детерминированной задержки.

Прием данных от АЦП осуществлялся согласно схеме (рисунок 1) с помощью логики, собранной на стороне СнК на основе стандарта JESD204B. Схема (как и стандарт) представлена тремя функциональными уровнями (слоями) организации взаимодействия между конвертером и обработчиком (в данном случае ПЛИС приемника): «Физический», «Линковочный» («Канальный») и «Транспортный».

3. Блок-дизайн JESD204B

Для реализации всех трех уровней стандарта JESD204B были использованы готовые IP-ядра от Analog Devices, доступные из открытого репозитория GitHub [4, 5]. Сборка этих ядер может отличаться в зависимости от конфигурации всей системы: необходимости в синхронизации; потребности в детерминированной задержке; количества линий данных и прочих параметров. Для реализации вышеупомянутой схемы был собран блок-дизайн, представленный на рисунке 2. Он включает 5 IP-ядер, объединенных в структуры в соответствии с «послойным» строением стандарта. Использование этих IP-ядер позволяет быстро адаптировать конфигурацию интерфейса на основе стандарта JESD204B для решения поставленных задач. Управление готовыми ядрами осуществляется через работу с их регистрами по интерфейсу AXI.

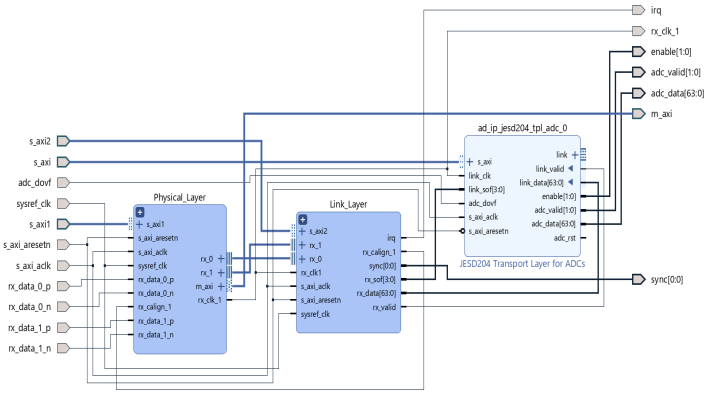


Рисунок 2. Общий блок-дизайн интерфейса JESD204B.

На вход в схему JESD204B подаются два дифференциальных канала данных (по одному от каждой компоненты сдвоенного АЦП) – *rx_data_0_p/n* и *rx_data_1_p/n*; интерфейсы AXI для взаимодействия с каждым уровнем стандарта – *s_axi_0/1/2*, *s_axi_resetn*, *s_axi_aclk*; тактовый сигнал – *sysref_clk*; и статусная линия, сигнализирующая о переполнении принимающего буфера – *adc_dovf*. На выходе из схемы имеются общая шина данных от обоих АЦП – *adc_data*, пара статусных шин (о готовности каждого из каналов обработки – *enable,* и валидности данных на каждом из них - *valid*), тактовый сигнал функционирования всего интерфейса, уменьшенный в два раза по частоте по сравнению с входным тактовым сигналом – *rx_clk_1*; линия прерывания – *irq*; и интерфейс AXI – *m_axi*.

Задачей физического уровня (рисунок 3) является взаимодействие с физической средой линии передачи данных и настройка высокоскоростных последовательных трансиверов [3] для обеспечения приема или передачи данных по среде распространения сигнала. Данный слой представлен в виде комбинации двух IP-ядер, один из которых предназначен для управление физическим уровнем протокола передачи, а второй для непосредственной конфигурации трансиверов ПЛИС в заданный режим работы в соответствии со стандартом. При переходе на линковочный уровень организуются две шины данных (*rx_0/1*) с понижением частоты тактового сигнала (*rx_clk_1*) в два раза по сравнению с рабочей частотой АЦП (до 125 МГц).

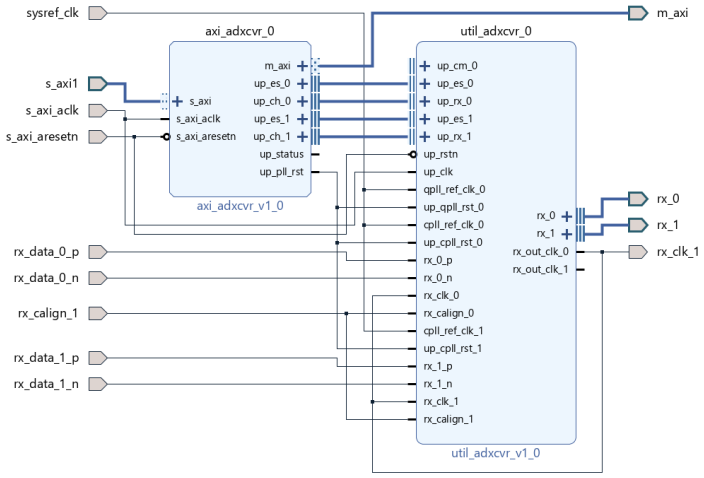


Рисунок 3. Блок-дизайн «Физического» уровня JESD204B.

Линковочный уровень (рисунок 4) включает в свою зону ответственности обработку (конвертацию) данных и контроль передачи данных. К нему в зависимости от версии стандарта может относится различный функционал, такой как кодирование и декодирование данных (по коду 8B/10B или 64B/66B), скремблирование, перестановка символов, обнаружение ошибок, проверки целостности данных, обработка состояния линий, синхронизация линий и работы преобразователей.

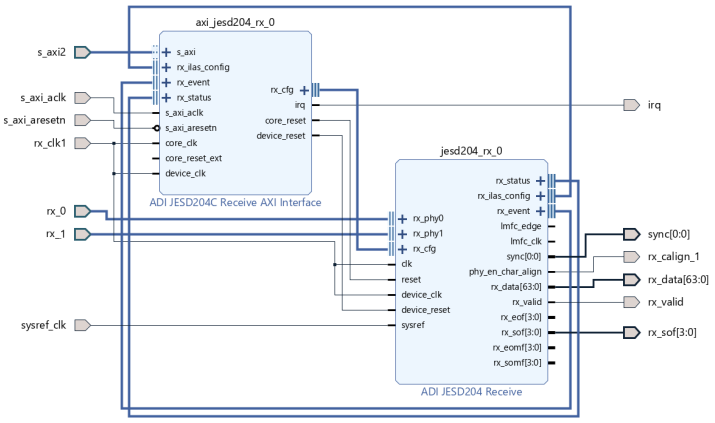


Рисунок 4. Блок-дизайн «Линковочного» уровня JESD204B.

Структурно данный уровень, как и физический, представлен двумя IP-ядрами: интерфейсным и функциональным. Соответственно, задачей одного из них является обеспечение взаимодействия с уровнем линковки посредствам интерфейса AXI, задачей второго – контроль обработки и передачи данных между уровнями: выявление ошибок в передаче; декодирование данных (8B/10B); их форматирование и объединение в общую шину; информирование о состоянии передачи и взаимодействие с соседними уровнями.

Транспортный уровень (рисунок 5) интерфейса JESD204B от Analog Devices представлен одним блоком. Для конечного пользователя основная функция данного блока заключается в переформатировании поступающих данных согласно заданным шаблонам. Фактически, он обеспечивает преобразование оцифрованных данных в четкую хронологическую последовательность в соответствии с конфигурацией всей системы конвертации. Кроме того, в данном блоке предусмотрен функционал управления работой каждой линией. Этот уровень позволяет оградить внутреннюю структуру и особенности протокола JESD204 от внешнего по отношению к нему пользователя, который нацелен на непосредственную работу с данными АЦП.

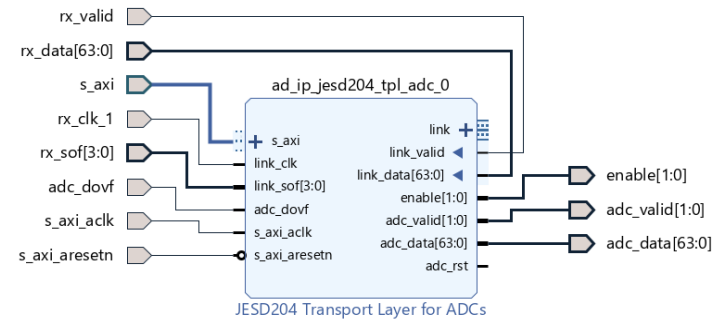
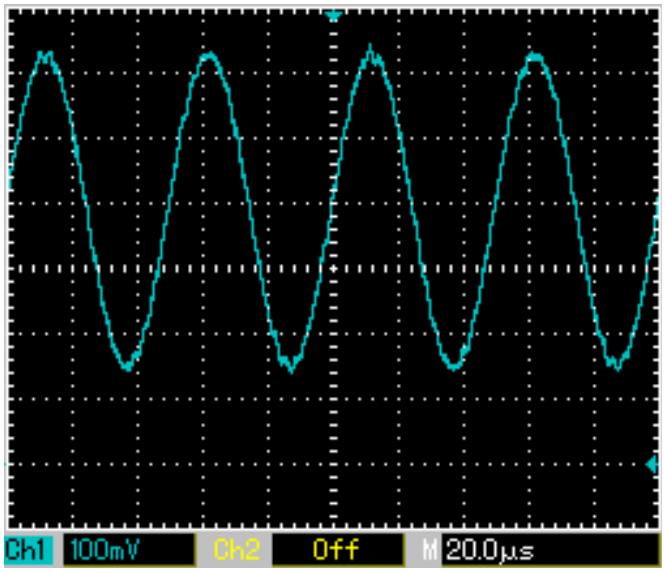


Рисунок 5. Блок-дизайн «Транспортного» уровня JESD204B.

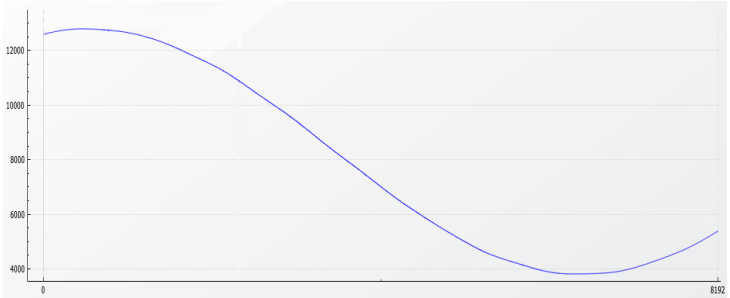
Блок обработки, указанный на схеме, представляет собой небольшой набор IP-ядер для сбора и записи данных в DDR память. На одном из процессоров СнК был запущен TCP-сервер, собранный на основе шаблонного проекта, под управлением FreeRTOS, что позволило обеспечить передачу по Ethernet собранных в памяти данных АЦП.

4. Эксперимент

Для проверки работоспособности разработанного стенда на каждый из каналов АЦП через драйверы были поданы синусоидальные сигналы от генератора UNI-T UTG2025A. Частота генерируемой синусоиды на одной из линий была установлена в значение 20 кГц (рисунок 6-а), другой 25 МГц (рисунок 7-а). Переданные по Ethernet данные регистрировались клиентской программой на сторонней рабочей станции, размещенной в одной сети с разработанным стендом.



а)



б)

Рисунок 6. Измерение 20кГц синусоидального сигнала
а) осциллографом; б) разработанной измерительной системой.

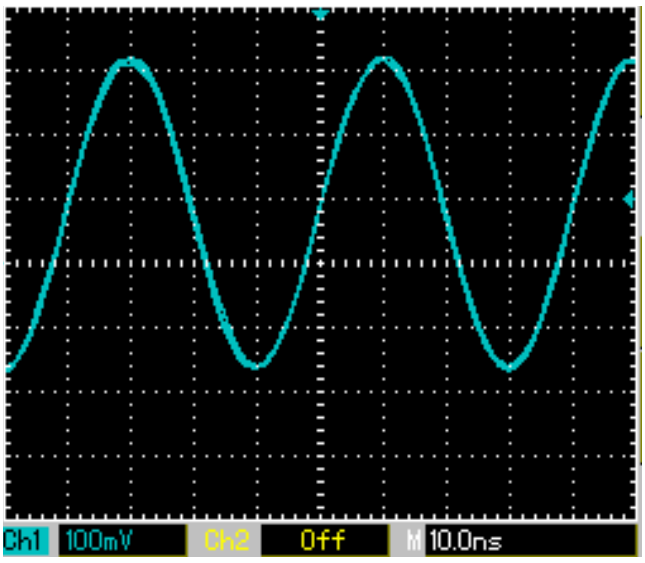
На рисунке 6-б представлены зарегистрированные данные от АЦП за 8192 такта работы, что соответствует записи продолжительностью 8192 * (1 / 250 МГц) = 32,768 мкс или приблизительно 2/3 периода заданной синусоиды.

На рисунке 7-б представлены отмасштабированные до

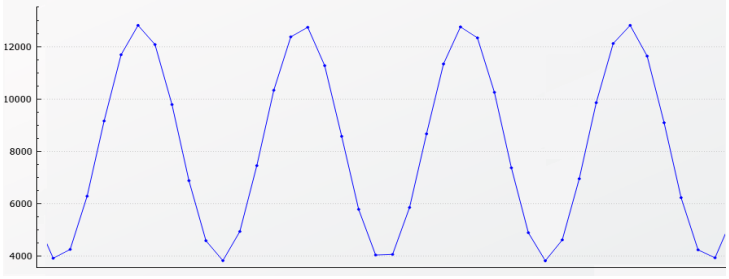
40 точек данные от АЦП за 8192 такта работы. Исходя из отношения частоты оцифровки (250 МГц) и частоты генерируемой синусоиды (25 МГц), в период зарегистрированной синусоиды должно укладываться 10 точек на графике. Данное следствие постановки эксперимента выполнялось на всей продолжительности записи, сделанной с помощью разработанной измерительной системы.

Также на рисунке наблюдается смещение точек, по которым восстанавливается синусоида, на каждом из периодов. Этот эффект может оказаться следствием либо дрожания сгенерированного сигнала, либо малым, но существенным отклонением рабочей частоты АЦП (сгенерированной тактовым генератором) от 250 МГц. На момент проведения испытаний проверить хотя бы одну из данных гипотез не представлялось возможным.

Тем не менее работоспособность системы оцифровки на частоте 250 МГц и последовательной передачи полученных данных со скоростью 5 Гбит/с (по интерфейсу на основе JESD204B) была полностью подтверждена экспериментально.



а)



б)

Рисунок 7. Измерение 25МГц синусоидального сигнала
а) осциллографом; б) разработанной измерительной системой.

5. Результаты

Результат выполненной работы заключается в реализации и тестировании гигабитного последовательного интерфейса на основе стандарта JESD204B. В процессе развития работы была достигнута стабильная скорость передачи данных 5 Гбит/с (между АЦП и СнК) при частоте оцифровки 250 МГц и разрядности АЦП 14 бит.

Выводы, полученные по ходу выполнения работы, указывают на то, что разработка дизайна ПЛИС с использованием IP-ядер из открытого репозитория позволяет значительно упростить запуск последовательного интерфейса, не требуя при этом глубокого изучения особенностей стандарта. Кроме того, на уровне проектирования дизайна ПЛИС значительно упрощается масштабирование всей системы регистрации и обработки данных. При таком подходе основное время развития проекта будет тратиться на реализацию и отладку системы обработки данных, полученных при помощи гигабитного последовательного интерфейса, так же, как и в случае работы с параллельной передачей.

Однако, использования стандарта JESD204B сопряжено с некоторыми ограничениями, связанными с наличием короткой, но потенциально критичной задержки между моментом регистрации данных и получением возможности их обработки (в сравнении с параллельными интерфейсами), а также необходимостью выделения части ресурсов ПЛИС под логику, обеспечивающую функционирование интерфейса.

Литература

1. Harris J. - What Is JESD204 and Why Should We Pay Attention to It?
2. JEDEC Standard JESD204C (December 2017). JEDEC Solid State Technology Association. www.jedec.org.
3. 7 Series FPGAs GTX/GTH Transceivers User Guide (UG476)
4. Репозиторий GitHub: https://github.com/analogdevicesinc/hdl
5. JESD204 Interface Framework: https://wiki.analog.com/resources/fpga /peripherals/jesd204

Аппаратная реализация на ПЛИС свёрточных нейронных сетей для семантической сегментации снимков леса

Мыцко Е.А., *evgenvt@tpu.ru*
Telegram: @evgenmytsko

Обсуждение и комментарии :: ссылка

Введение

В настоящее время в мире актуальными являются задачи распознавания различных объектов (лица людей, автомобили, объекты земной поверхности и.т.д) и мониторинга их состояния. Для решения таких задач успешно используются сверточные нейронные сети (СНС) различных классов. Однако, при решении некоторых из таких задач возникает потребность в использовании беспилотных летательных аппаратов (БПЛА) с установленным на них специальным оборудованием для мониторинга труднодоступных объектов (опасные технологические объекты на промыслах, лесные массивы в горных районах и т.п.). В таких случаях для мониторинга требуется оснащать каждый БПЛА интеллектуальной системой компьютерного зрения (СКЗ), которая будет включать видеокамеру, тепловизор и вычислительное устройство (ВУ) с аппаратно-реализованной СНС, позволяющие прямо на борту беспилотного аппарата решать задачи распознавания объектов различной физической природы (автомобили, технологические объекты, люди и т. д.). При проектировании таких мобильных систем мониторинга с интеллектуальными СКЗ необходимо соблюдать баланс между быстродействием используемого ВУ, точностью распознавания объектов земной поверхности с помощью СНС, массой и энергопотреблением этого устройства с целью увеличения времени полёта БПЛА.

Данная работа посвящена исследованию эффективности аппаратно-реализованных моделей СНС класса U-net на программируемых логических интегральных схемах (ПЛИС) современных систем на кристалле (СнК). Для обучения, верификации и исследования таких моделей СНС применялся датасет, созданный с использованием снимков с БПЛА деревьев пихты сибирской, поврежденных уссурийским полиграфом. В зависимости от степени повреждения вредителем на снимках присутствуют деревья четырех классов пихты и фон. Таким образом, в работе продемонстрированы результаты по быстродействию, точности сегментации изображений деревьев, а также энергопотреблению аппаратной реализации моделей СНС класса U-net в составе ВУ.

Задача семантической сегментации снимков леса

В работе ставится задача семантической сегментации снимков с БПЛА деревьев пихты сибирской, поврежденных уссурийским полиграфом. В зависимости от степени повреждения вредителем на снимках присутствуют деревья четырех классов («живые», «отмирающие», «свежий сухостой» и «старый сухостой»), а также фон. Предлагаемые для решения этой задачи модели СНС класса U-Net разработаны на основе классической полносверточной сети U-Net. Классическая СНС этого класса состоит из энкодера и декодера. Характерной особенностью моделей СНС класса U-Net является наличие операций конкатенации, соединяющих карты признаков из энкодера с картами признаков в декодере с целью повышения детальности результирующих карт классификации.

Классическая модель СНС U-Net была модифицирована с учетом необходимости классификации пикселей, соответствующих четырем классам деревьев пихты и фону. На рис. 1 представлена архитектура разработанной модели СНС, где каждому прямоугольнику соответствует тензор – многомерный массив, представляющий собой набор карт признаков, числами показано число компонентов тензоров.

Стрелками показаны соответствующие операции: свертка (Conv3x3, Conv1x1), вычисление функции активации leakyReLU (использовались также функции ReLU и eLU), пакетная нормализация (BN), уменьшение масштаба карт признаков путем выбора максимального значения в окрестности 2x2 в картах признаков (MaxPooling), операция увеличения масштаба карт признаков методом ближайшего соседа (UpSampling), копирование тензора и его конкатенация с другим (Copying+Concatenation) и исключение случайных сигналов слоя путем приравнивания их значений к нулю

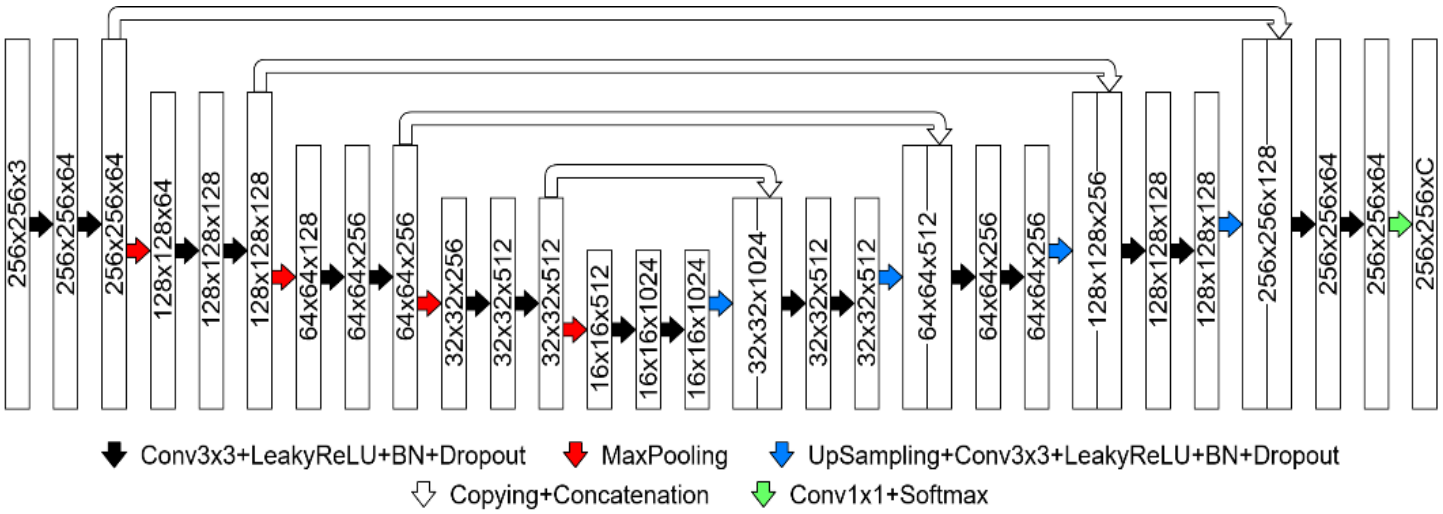


Рис.1 - Архитектура модели СНС, разработанной на основе классической U-Net

(Dropout). Категориальное распределение на выходе декодера моделируется для каждого пикселя путем применения многомерной логистической функции Softmax. В отличие от классической U-Net эта модель СНС обладает следующими особенностями: входное изображение сети представлено тензором с числом компонентов 256x256x3, что соответствует фрагменту трехканального RGB снимка; операции сверток не уменьшают размер выходных карт признаков; обрезка карт признаков не используется для соединений проброса; после каждого вычисления функции leakyReLU следует операция пакетной нормализации; выходной тензор формируется с помощью сверток с ядрами размером 1x1, позволяя тем самым сразу классифицировать пиксели С классов деревьев пихты и фон.

Для оценки эффективности предложенной модели СНС при проведении их исследований использовалась метрика intersection over union (*IoU*), считающаяся общепринятой метрикой эффективности при решении задачи семантической сегментации цифровых изображений. Известно, что значения *IoU*, превышающие 0,5, соответствуют приемлемому качеству сегментации. В качестве совокупного показателя качества модели использовалась метрика mean intersection over union (*mIoU*), рассчитываемая как среднее значение *IoU* по всем классам деревьев пихты и фону.

Аппаратная реализация предложенной модели СНС

Известно, что современные мобильные СКЗ, устанавливаемые на автономных транспортных средствах, в том числе на БПЛА, должны иметь малое энергопотребление. Например, СКЗ на БПЛА должны потреблять не более 10--12 Вт. Поэтому ВУ таких СКЗ следует создавать на основе современных СнК с ПЛИС, обладающих малым энергопотреблением и высокой производительностью. Нами использовалась СнК Zinq 7000 (Kintex FPGA) компании Xilinx.

Архитектура большинства СнК позволяет организовать прямой доступ аппаратно-реализованной на ПЛИС модели СНС к внешней памяти. Такая организация взаимодействия ПЛИС с внешней памятью позволяет выполнять некоторые операции, отличные от процедур свёртки и подвыборки СНС, не на ПЛИС, а на процессоре (процессорной системе) СнК. В работе предложен оригинальный метод выполнения вычислений в аппаратной СНС на ПЛИС, отличающийся от известных методов использованием унифицированных вычислительных блоков свёртки и подвыборки. Унификация блоков свёртки/подвыборки достигается путем извлечения параметров блоков, обычно задающихся на этапе их синтеза, и размещения их в отдельную изменяемую область памяти ПЛИС, называемую конфигурационной областью памяти CONFIG space. Это позволяет использовать блоки в слоях СНС, имеющих разные архитектурные параметры. Более того, в слоях свертки и подвыборки аппаратной СНС используются блоки только одного типа – универсальные блоки. Такие блоки содержат как субблоки свертки, так и субблоки подвыборки. Это позволяет более гибко организовать вычисления в аппаратной СНС. Реализация метода предполагает, что число задействованных в аппаратной СНС универсальных вычислительных блоков может быть переменным и определяется только ресурсами ПЛИС.

При реализации метода унификации кроме выделения конфигурационной области памяти CONFIG space необходимо было организовать доступ к этой области памяти процессору СнК, для чего было создано соответствующее программное обеспечение.

Унификация универсальных вычислительных блоков аппаратной СНС и предложенный в способ организации вычислений в СнК с учетом необходимости взаимодействия ПЛИС с внешней памятью позволили реализовать ВУ на основе отладочной платы Avnet Mini-ITX Board с СнК Zynq 7000 (Kintex FPGA). Укрупненная функциональная схема этого ВУ представлена на рисунке 2.

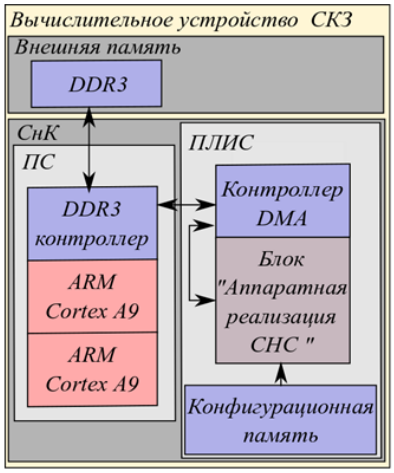


Рис. 2. Укрупненная функциональная схема ВУ

В устройстве (рис. 2) два основных функциональных блока. Первый из них — СнК, он содержит процессорную систему (ПС) и ПЛИС. В состав ПС, в свою очередь, входят контроллер внешней памяти DDR3-типа и два процессорных ядра ARM Cortex A9 с тактовой частотой 800 МГц. На ПЛИС созданы контроллер прямого доступа к внешней памяти (Контроллер DMA), блок «Аппаратная реализация СНС» и конфигурационная память CONFIG space, входящая в этот блок (на рис. 2 конфигурационная память для наглядности представления схемы ВУ вынесена за пределы блока). В блоке «Аппаратная реализация СНС» имеется также нейровычислительное устройство (НВУ, рис. 3), которое и выполняет вычисление процедур свертки и подвыборки для моделей СНС класса U-Net. НВУ содержит 64 универсальных вычислительных блока, которые одновременно работают над каждым слоем СНС. Вторым функциональным блоком ВУ является внешняя память DDR3 с эффективной частотой 800 МГц.

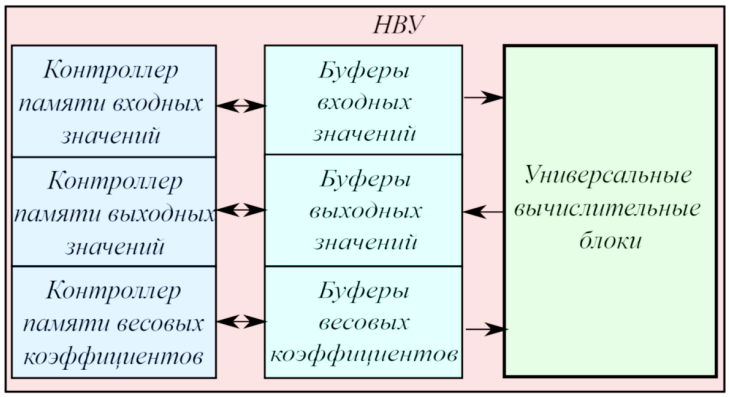


Рис. 3. Укрупненная функциональная схема НВУ в составе ВУ

В его состав также входят контроллеры памяти и буферы входных/выходных значений и весовых коэффициентов слоев СНС.

Функциональная схема отдельного универсального вычислительного блока НВУ представлена на рис.4. Блок состоит из трех конечных автоматов (два входных input_st_machine 1, input_st_machine 2 и один выходной output_st_machine), четырех мульти-плексоров 2-1 (Mux 1-4), сумматора (sum), умножителя (mult) и компаратора (comp).

Верхний входной конечный автомат (input_st_machine 1) работает с входными данными для слоя (изображение на входе первого слоя СНС или карта признаков, подаваемая на вход последующего слоя), а нижний (input_st_machine 2) – с весовыми коэффициентами слоя свертки СНС. Два последовательно подключенных мультиплексора (mux 1 и mux 2, mux 3 и mux 4) реализуют процедуру подвыборки. Данные от первого

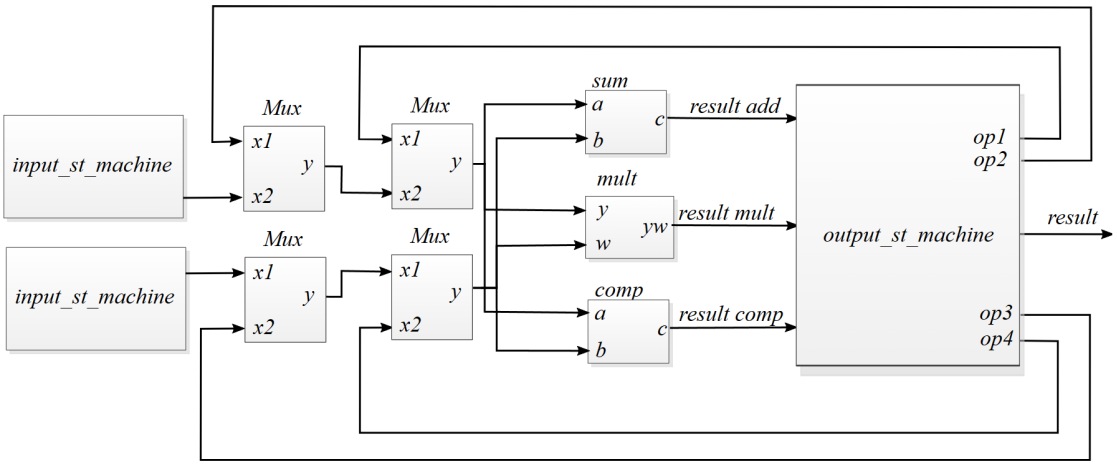


Рис. 4. Функциональная схема универсального вычислительного блока

входного конечного автомата поступают на один из входов первого мультиплексора. В зависимости от состояния выходного конечного автомата эти данные поступают либо на выход первого мультиплексора, либо на выход мультиплексора поступают данные с предыдущей итерации вычислений (op2). На втором мультиплексоре происходят аналогичные процессы, только одним из входов является выход первого мультиплексора. Вторым входом mux 2 является результат вычислений на предыдущей итерации (op1). Выходы мультиплексоров 2 и 4 подключены к входам сумматора (sum), умножителя (mult) и компаратора (comp).

Нижний входной конечный автомат (input_st_machine 2) работает с весовыми коэффициентами слоя свертки. Процедура подвыборки аналогична процедуре для входных данных изображения. Обратные связи от выходных сигналов op3 и op4 позволяют реализовать итеративный процесс обработки данных. Сумматор и умножитель итеративно выполняют процедуру свертки на основе входных данных изображения и весовых коэффициентов слоя свертки. Компаратор реализует функцию активации. Таким образом, на выходе result блока будет сформирован результат вычисления одного слоя СНС.

Изменяя количество используемых универсальных вычислительных блоков в НВУ можно получить некоторый баланс между быстродействием ВУ мобильной СКЗ и аппаратными ресурсами ПЛИС, задействованными при реализации модели СНС. Таким образом, предложен и реализован алгоритм работы НВУ,в соответствии с которым ведутся параллельные вычисления внутри каждого слоя модели СНС. Идея алгоритма заключается в том, что каждый универсальный вычислительный блок НВУ производит последовательное вычисление элементов назначенной ему выходной карты признаков в текущем слое. Получается, что каждая выходная карта признаков одновременно формируется отдельным таким блоком. Если выходных карт признаков слоя больше количества вычислительных блоков в НВУ, то один блок производит поочередное формирование нескольких выходных карт признаков слоя.

Для обеспечения взаимодействия ПЛИС, ПС и внешней памяти разработано программное обеспечение в виде многофункционального драйвера под ОС с ядром Linux. Драйвер написан на языке Си. На языке C++ разработана библиотека для взаимодействия пользователя с ВУ при его настройках.

Для проведения быстрого реконфигурирования НВУ на ПЛИС и с целью упрощения аппаратной реализации различных моделей СНС был разработан программный инструмент на языке Python с процедурами (вставками) на языке Cython. Этот инструмент также дает возможность выбора варианта реализации модели СНС на процессорной системе (x86 или ARM) или на ПЛИС.

Исследование аппаратно-реализованных моделей СНС

Для исследования эффективности аппаратно-реализованных моделей СНС в составе ВУ проведены эксперименты по семантической сегментации фрагментов изображения с БПЛА участка деревьев пихты. Фрагменты изображения размером 256x256x3 подавались на вход аппаратно-реализованную в ВУ модель СНС U-Net (предварительно обучена программно).

Эксперименты с моделью СНС U-Net проводились как с использованием 16-разрядных (float16), так и 32-разрядных (float32) чисел с плавающей запятой и с разными функциями активации: eLU, ReLU и leakyReLU. Результаты для этой модели приведены в таблице 1.

Таблица 1. Значения точности семантической сегментации IoU фрагментов, полученные с помощью модели СНС U-Net

Функция активации	Точность семантической сегментации *IoU* для различных классов деревьев пихты и фона					
	Фон	Живые	Отмирающие	Свежий сухостой	Старый сухостой	mIOU
float16						
eLU	0.84	0.72	0.39	0.74	0.63	0.66
ReLU	0.82	0.68	0.28	0.76	0.66	0.64
leakyReLU	0.84	0.72	0.36	0.77	0.65	0.67
float32						
eLU	0.84	0.72	0.39	0.74	0.63	0.66
ReLU	0.83	0.68	0.29	0.76	0.66	0.64
leakyReLU	0.84	0.72	0.36	0.77	0.65	0.67

Видим, что лучшие результаты по точности сегментации фрагментов получены при использовании модели СНС U-Net с функцией активации leakyReLU. Для дальнейших исследований эффективности аппаратно-реализованных моделей СНС использовались результаты обучения в виде весовых коэффициентов программно-реализованных моделей СНС только с этой функцией активации.

Исследование скорости вычисления моделей СНС при их аппаратной реализации на ПЛИС заключалось в измерении времени, затраченного на семантическую сегментацию одного фрагмента размером 256x256x3 на ПЛИС. СнК позволяла работать ПЛИС с тактовой

частотой 100 МГц и организовать использование в НВУ 64 универсальных вычислительных блоков. Результаты по среднему времени обработки по всем используемым в экспериментах фрагментам для случаев float32 и float16 приведены в таблице 2.

Таблица 2. Скорость вычисления моделей СНС на ПЛИС

CNN model	Среднее время обработки фрагмента, с			
	float32		float16	
	Медиана	MAD	Медиана	MAD
U-Net	31.12	0.07	38.03	0.12

Использование чисел float16 («IEEE float16») приводит к падению производительности ВУ из-за того, что слои upsample и последний слой CNN обрабатываются на ПС, которая не поддерживает числа с плавающей запятой половинной точности. Преобразование чисел из формата float32 в float16 требует дополнительных затрат, что является узким местом ВУ.

В таблице 3 приведены результаты измерения энергопотребления СнК и требующихся ресурсов ПЛИС в виде количества различных ячеек при аппаратной реализации каждой из моделей СНС U-Net в случаях использования чисел в форматах float32 и float16 .Анализируя эти результаты, можно сказать, что применение чисел с плавающей запятой float16 позволяет уменьшить по сравнению со случаем чисел float32 энергопотребление СнК и требуемые ресурсы ПЛИС для хранения входных данных и весовых коэффициентов модели СНС. Энергопотребление СнК с ПЛИС составляет немного более 4-5 Вт, что в десятки раз меньше, чем у того же графического ускорителя NVIDIA GeForce GTX 1080 Ti, имеющего энергопотребление 250 Вт.

Таблица 3. Энергопотребление СнКи ресурсы ПЛИС

Энергопотребление и ресурсы ПЛИС				
Тип	Энерго-потребление, Вт	LUT, шт	LUTRAM,	BRAM, шт
float16	4.32	138917	2074	225,5
float32	5.33	165278	871	328,5

Все эти результаты еще раз указывают на то, что основным преимуществом устройств на основе современных СнК с ПЛИС является низкое энергопотребление. Более того, современные СнК обладают весьма малой массой по сравнению с теми же графическими ускорителями.

Отметим, что все полученные результаты исследования аппаратно-реализованных на ПЛИС двух моделей СНС класса U-Net являются весьма важными для разработчиков интеллектуальных СКЗ в составе мобильных систем мониторинга на основе БПЛА. Разработчики таких СКЗ должны соблюдать баланс между быстродействием используемого ВУ, точностью распознавания объектов земной поверхности с помощью СНС, массой и энергопотреблением этого устройства с целью увеличения времени полёта БПЛА. При поиске баланса им достаточно проанализировать приведенные результаты исследований и принять соответствующие проектные решения.

Заключение

Сегодня актуальными являются задачи создания интеллектуальных СКЗ в составе мобильных систем мониторинга объектов земной поверхности на основе БПЛА. В данной работе с целью последующей разработки такой СКЗ предложены, программно-реализованы и обучены две новые модели СНС класса U-net. Затем эти модели СНС аппаратно-реализованы на ПЛИС современной СнК Zynq 7000 (Kintex FPGA). Для обучения, верификации и исследования моделей СНС использовался датасет, созданный по снимкам с БПЛА деревьев пихты сибирской, поврежденных уссурийским полиграфом.

Полученные результаты измерения энергопотребления СнК и требующихся ресурсов ПЛИС при аппаратной реализации моделей СНС класса U-Net указывают на то, что применение 16-разрядных чисел с плавающей запятой при проведении вычислений моделей СНС позволяет уменьшить энергопотребление СнК и требуемые ресурсы ПЛИС по сравнению со случаем применения 32-разрядных чисел. Энергопотребление СнК с ПЛИС составляет немногим более 5 Вт, что в десятки раз меньше, чем у того же графического ускорителя NVIDIA GeForce GTX 1080 Ti, имеющего энергопотребление 250 Вт.

Можно считать, что полученные результаты исследований, предложенных аппаратно-реализованных на ПЛИС моделей СНС класса U-Net имеют важное научное и практическое значение для разработчиков интеллектуальных СКЗ в составе мобильных систем мониторинга объектов земной поверхности на основе БПЛА. Использование этих результатов позволит им принимать обоснованные проектные решения при создании интеллектуальных СКЗ для решения различных прикладных задач.

Об использовании фильтров в GTKWave

Пузанов Николай
Telegram @punzik

Обсуждение и комментарии :: ссылка

1. Введение

GTKWave является популярным открытым программным обеспечением для визуализации и анализа временных диаграмм, полученных от симуляторов RTL или от реальных устройств. Программа предлагает широкий спектр инструментов и функций, которые делают ее хорошим выбором как для начинающих разработчиков и студентов, так и для опытных инженеров. В данной статье мы рассмотрим применение фильтров в GTKWave и их использование для разбора и визуализации данных и транзакций.

В этой статье мы кратко рассмотрим работу с фильтрами трансляции и транзакционными фильтрами в GTKWave, а также разберем практические примеры их использования.

После прочтения этой статьи вы сможете самостоятельно написать фильтр и применить его в своей работе.

2. Translate Filter

Translate Filter - это простой фильтр, заменяющий одно значение на другое. На вход фильтра подаётся значение, взятое из диаграммы, а на выходе программа ожидает строку, которая будет использована для замены этого значения. Значение на вход подаётся в виде строки с теми же символами, который вы видите на диаграмме. Т.е. если на диаграмме сигнал отображается в виде шестнадцатеричной строки, но на вход фильтра придёт именно эта строка с шестнадцатеричными символами.

2.1 Translate Filter File

Файловый транслятор - это самый простой фильтр, который представляет собой файл с двумя колонками, разделенными пробелом. В первой колонке строка, которую нужно заменить (ключ), вторая колонка - строка замены.

Например, у нас есть конечный автомат с такими состояниями:

```
enum int unsigned {
    ST_IDLE = 0,
    ST_CHECK_ADDR,
    ST_RECEIVE_DATA,
    ST_SEND_ACK,
    ST_DONE
} state;
```

Если установить формат данных Decimal, то на диаграмме это будет выглядеть так, как показано на рис.1.

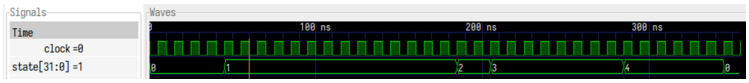


Рис.1 Временная диаграмма состояний конечного автомата

Создадим файл фильтра `state-filter.txt`:

```
0 IDLE
1 CHECK_ADDR
2 RECEIVE_DATA
3 SEND_ACK
4 DONE
```

А теперь применим его для сигнала `state` и посмотрим результат. Нужно отметить не очень интуитивное поведение диалога добавления фильтра, которое в первый раз может вызвать непонимание. Для применения фильтра нужно правой кнопкой щёлкнуть на сигнал и в меню выбрать "Data Format/Translate Filter File/Enable and Select" (то же самое можно сделать через меню "Edit"). В открывшемся диалоговом окне добавить в список нужный файл кнопкой "Add Filter to List" (если он там уже есть, добавлять не нужно), а затем выбрать его в списке и нажать "OK". Если файл в списке не будет выбран, фильтр не будет применен к сигналу.

Итак, после применения фильтра на временной диаграмме вы увидите названия состояний вместо их номеров, как показано на рис.2.

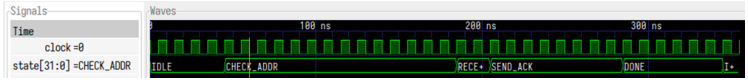


Рис.2 Диаграмма с именованными состояниями автомата

Если изменить формат представления данных на HEX, то значения будут отображаться в виде шестнадцатеричных строк (например, `00000001`), потому что таких ключевых строк в фильтре нет. Если вернуть обратно десятичный формат, то значения снова будут отображаться в виде названий состояний автомата.

Для удобства программа ищет ключевые строки в файле без учёта регистра. Выходной строкой считается всё, что находится после ключа и до конца строки, исключая пробелы в начале и в конце. Т.е. в качестве строки замены можно использовать строку с пробелами или другими знаками.

Поиск осуществляется с первой по последнюю строку до совпадения ключа. Соответственно, если в файле несколько строк с одинаковым ключом, будет выбрана первая.

Кроме подстановки текста, программа позволяет раскрасить фон диаграммы. Для этого перед строкой замены нужно указать название цвета, обрамленное в знаки вопроса. Например, если заменить строки IDLE и RECEIE_DATA на показанные ниже, то соответствующие участки диаграммы раскрасятся в указанные цвета (рис.3).

```
0 ?blue violet?IDLE
2 ?brown4?RECEIVE_DATA
```

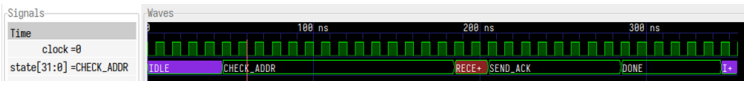


Рис.3 Диаграмма с цветными состояниями

В палитре больше 700 цветов. Посмотреть названия их всех можно в исходниках в файле `src/rgb.c` (в стабильной версии) или в файле `lib/libgtkwave/src/gw-color.c` (https://github.com/gtkwave/gtkwave/blob/master/lib/libgtkwave/src/gw-color.c) в последней версии на день написания статьи.

2.2 Translate Filter Process

Если простого текстового фильтра не хватает, например, если нужно выполнять какие-то вычисления, то можно использовать фильтр на основе внешней программы или скрипта. Такой фильтр называется "Translate Filter Process".

Принцип простой: GTKWave запускает внешнюю программу и при каждой перерисовке диаграммы запрашивает у неё замены для отображаемых значений. Запросы поступают на стандартный вход в виде строк, оканчивающихся символом перевода строки, а со стандартного вывода ожидает ответ в виде строки для замены. На каждый запрос должен быть отправлен ответ. Важно отметить, что после ответа программа должна принудительно сбросить буферы вывода (flush), иначе GTKWave зависнет в ожиданиии ответа.

Запросы в программу поступают при каждой перерисовке диаграммы. По завершении работы GTKWave закрывает исходящий поток, а программа получает код "End Of File".

Формат выходной строки такой же, как в текстовом фильтре. Это касается и способа раскраски фона.

Для примера попробуем раскрасить диаграмму исходя из значения знакового вектора. Если значение меньше нуля - синий, если от нуля до трёх, то оставим как есть, если 4 и больше - коричневый. А если это не десятичное число, то выведем строку "NaN" на красном фоне.

Вот скрипт на питоне, который выполняет такую фильтрацию:

```python
#!/usr/bin/env python

import sys

for line in sys.stdin:
    try:
        key_value = int(line)

        if key_value < 0:
            print("?blue4?{}".format(key_value))
        elif key_value < 4:
            print(key_value)
        else:
            print("?brown4?{}".format(key_value))

    except ValueError:
        print("?dark red?NaN")

    sys.stdout.flush()
```

Снова напомню, что после каждой строки необходимо сбрасывать буфер - `sys.stdout.flush()`.

После применения фильтра мы получим следующую картинку, показанную на рис.4 (на диаграмме два одинаковых сигнала - верхний без фильтра, нижний - с фильтром).

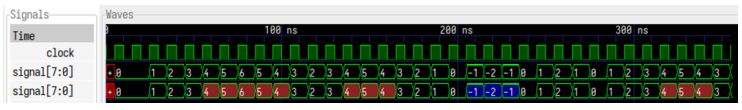


Рис.4 Фильтрация с помощью Translate Filter Process

И опять, как и в предыдущем случае, чтобы всё работало корректно, необходимо установить формат сигнала в "Signed Decimal", т.к. на вход фильтра подаются строки в таком виде, в котором они отображаются на диаграмме.

3. Transaction Filter

Фильтры, описанные выше, имеют один существенный недостаток - с их помощью невозможно отследить изменения сигнала во времени, что не позволяет, например, разобрать протокол передачи по шине и вывести на диаграмму информацию о транзакциях.

Для разбора транзакций в GTKWave добавили ещё один тип фильтров, который называется Transaction Filter Process. Работает он почти так же, как Translate Filter Process, но принимает на вход упрощенный дамп VCD с нужными сигналами, и возвращает некое подобие VCD с новыми сигналами, которые будут добавлены на диаграмму. Т.е. фильтр имеет доступ ко всей и истории, позволяя разобрать как отдельные значения сигналов, так и транзакции.

3.1 UART

В качестве первого примера сделаем разбор протокола UART. Это достаточно простой последовательный протокол, использующий всего один сигнал для передачи данных.

На рис.5 показана диаграмма с двумя сигнала передачи UART на разных скоростях. Формат передачи: 8 бит, 1 стоп-бит, без контроля чётности. Попробуем определить скорость передачи и узнать, что было передано.

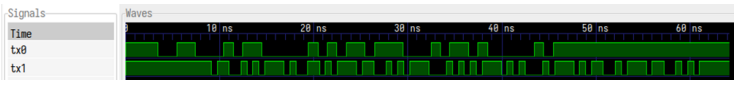


Рис.5 Две передачи по протоколу UART

Итак, как было сказано выше, GTKWave передаёт на вход фильтра дамп VCD с сигналами, для которых был применён фильтр. Формат дампа соответствует стандарту VCD, но для упрощения тэги в нём не переносятся на новую строку, т.е. при разборе можно не искать закрывающий тэг `$end`, а полагаться только на открывающий. Кроме того, строки не могут содержать пробелов в начале и в конце, а между словами только один пробел, что тоже упрощает разбор.

GTKWave присылает дамп в виде блоков, начинающихся с комментария `data_start` и заканчивающихся комментарием `data_end`. Шестнадцатеричное число внутри комментария - это уникальный идентификатор блока, который не используется при фильтрации и предназначен только для отладки (на самом деле это значение указателя на структуру внутри программы).

```
$comment data_start 0x39f9a40 $end
... тело запроса ...
$comment data_end 0x39f9a40 $end
```

В блоке находятся данные о состояниях сигналов, для которых был применен фильтр. При этом, если просто применить фильтр для нескольких сигналов, то эти сигналы будут переданы по отдельности, каждый в своём блоке. Если нужно разбирать шину, состоящую из нескольких сигналов, то эти сигналы нужно сначала объединить с помощью команды "Edit/Combine Down", а затем уже применять к ним фильтр.

После получения каждого такого блока необходимо сформировать ответ, содержащий список новых сигналов с их значениями. Список состоит из блоков, начинающихся с тэга `$name` и заканчивающихся тегом `$next`, если блок не последний, или `$finish`, если в списке больше нет сигналов.

```
$name signal0_name
...
$next
$name signal1_name
...
$finish
```

Сделаем каркас для фильтра и выведем в поток стандартного вывода ошибок (`stderr`) дамп того, что нам прислал GTKWave. Так мы сможем своими глазами увидеть формат запроса, поступающего от GTKWave. Чтобы GTKWave не зависла, сформируем и выведем в поток стандартного вывода минимальный ответ. Без этого программа зависнет и её придется принудительно перезагружать.

Максимально упростим разбор и будем определять границу блока только по комментарию `data_end`.

```
#!/usr/bin/env python

import sys

# Версия print с принудительным сбросом буфера stdout
def pprint(*args, **kwargs):
    print(*args, **kwargs)
    sys.stdout.flush()

for line in sys.stdin:
    # Выводим дамп в stderr, чтобы посмотреть, что нам прислал GTKWave
    sys.stderr.write(line)

    # По окончанию приёма блока пошлём ответ
    if (line.startswith("$comment data_end")):
        pprint("$name New Signal")
        pprint("#0 ?dark cyan?Just Text")
        pprint("$finish")
```

Здесь стоит напомнить, что после вывода каждой строки необходимо сбрасывать выходной буфер. Для этого в коде используется функция `pprint`, которая делает то же самое, что `print`, но после вывода сбрасывает выходной буфер функцией `sys.stdout.flush()`.

Применим фильтр к сигналу, и посмотрим, что получилось на рис.6.

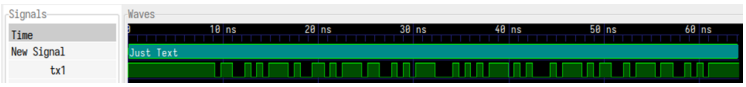


Рис.6 Результат применения простейшего транзакционного фильтра

Как видим, сигнал `tx0` был просто заменён на новый. Если из фильтра напечатать ещё один сигнал, то он отобразится ниже. Но предварительно необходимо добавить на диаграмму несколько пустых строк ("Insert Blank"), иначе новый сигнал некуда будет вставить. Если вы хотите, чтобы новый сигнал отображался рядом, а не заменял старый, можно вызвать команду "Combine Down", а затем уже применить фильтр.

В поток стандартных ошибок мы вывели дамп запроса от GTKWave. Вот он (показан не полностью):

```
$comment name tx0 $end
$timescale 1ps $end
$comment min_time 0 $end
$comment max_time 64134 $end
$comment max_seqn 1 $end
$scope module uart $end
$comment seqn 1 uart.tx0 $end
$var wire 1 1 tx0 $end
$upscope $end
$enddefinitions $end
#0
$dumpvars
11
$end
#3456
01
#5456
11
...
#44456
01
#45456
11
#64134
$comment data_end 0x1d95c10 $end
```

Сделаем минимальный разбор входного VCD и сложим значения сигнала с временными метками в массив в виде кортежей (tuple). Так же, извлечём имя сигнала с индексом 1, чтобы потом использовать его в имени нового сигнала.

```
sig_name = ""
sig_smpl = []
timestamp = 0

for line in sys.stdin:
    # Извлекаем имя сигнала с индексом 1
    if line.startswith("$comment seqn 1"):
        sig_name = line.split()[3]
    # Временная метка
    elif line[0] == '#':
        timestamp = int(line[1:])
    # Добавляем сигнал в массив
    elif (line[0] == '0' or line[0] == '1') and line[1]
== '1':
        sig_smpl.append((timestamp, int(line[0])))
```

После окончания приёма блока в массиве будут храниться пары с меткой времени и значением сигнала в этот момент.

После приёма блока выведем первую строку ответа с именем нового сигнала:

```
# По окончанию приёма блока пошлём ответ
if line.startswith("$comment data_end"):
    pprint("$name {} (flt)".format(sig_name))
```

Теперь нужно определить скорость передачи. Сделаем предположение, что в достаточно длинной посылке с высокой вероятностью найдется байт данных с комбинацией `101` или `010`. Длительность этого одинокого нуля или единицы примем за длительность бита. Т.е. в нашем массиве найдем переход от 0 к 1 или наоборот с минимальной длительностью.

```
# Найдем минимальное время между переключениями сигнала
t = sig_smpl[0][0]
s = sig_smpl[0][1]
dt = None # здесь будет длительность бита

for smpl in sig_smpl:
    if s != smpl[1]:
        dt_new = smpl[0] - t
        dt = dt_new if dt is None else min(dt, dt_new)
        t = smpl[0]
        s = smpl[1]
```

Остаётся найти первый стартовый бит (переход от 1 к 0) и относительно его середины прочитать значения следующих 8 бит. Это будет байт переданных данных.

```
smpl_idx = 0
t_start = None
tm = 0

while True:
    # Ищем стартовый бит
    start = get_following(tm, 0)
    if start is None:
        break

    # Ставим указатель времени на середину стартового
бита
    tm = start[0] + dt * 0.5
    b = 0

    # Читаем значение байта
    for n in range(8):
        tm = tm + dt
        b = (b >> 1) | (get_signal(tm) << 7)

    # Ищем стоповый бит
    stop = get_following(tm, 1)
    if stop is None:
        stop = sig_smpl[-1][0]
```

Примечание: код вспомогательных функций в конце статьи.

После получения байта можно сразу выдать ответ:

```
# Выводим сигнал
pprint("#{} {}".format(start[0], chr(b) if b >= 32 and b
<=127 else str(b)))
pprint("#{}".format(stop[0] + dt))
```

Первая строка возвращает время начала стартового бита и значение полученного байта (символ ASCII или код, если символ непечатный). Вторая строка печатает только время окончания стопового бита без значения.

Затем возвращаемся к поиску следующего стартового бита и повторяем процедуру, пока не закончатся элементы массива.

Т.к. новый сигнал один, после окончания разбора выведем строку `$finish` и обнулим массив сэмплов для разбора следующего блока:

```
pprint("$finish")
sig_smpl = []
```

Результат применения фильтра показан на рис.7.

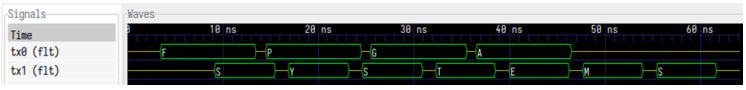


Рис.7 Результат применения транзакционного фильтра UART

Диаграммы сигналов были заменены на их отфильтрованные версии. Если сделать копию каждого сигнала через "Combine Down" и затем применить фильтры, то получим картину, показанную на рис.8.

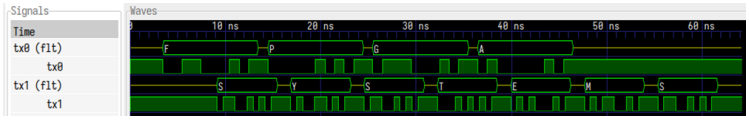


Рис 8. Транзакционный фильтр и Combine Down

И в заключение код вспомогательных функций:

```
# Возвращает значение сигнала в момент времени
def get_signal(tm):
    s = 0
    for smpl in sig_smpl:
        if smpl[0] <= tm:
            s = smpl[1]
        elif smpl[0] > tm:
            break

    return s

# Возвращает элемент массива sig_smpl, время которого >=
tm, а значение равно val
def get_following(tm, val):
    smpl = None
    for n,s in enumerate(sig_smpl):
        if s[0] >= tm and s[1] == val:
            smpl = s
            break

    return smpl
```

3.2 AXI

С помощью фильтра транзакций можно разбирать не только такие простые шины, как UART, но и сложные. Такие, например, как транзакции на шине AXI.

Для примера попробуем разобрать несколько транзакций в канале чтения на шине AXI4. Для наглядности, и чтобы не усложнять код фильтра, не будет рассматривать все сигналы, возьмём только часть.

На рис.9 изображена диаграмма с 6 транзакциями на шине AXI. Первые две со значением `arid` 0 и 1 - корректные транзакции чтения, ответы на которые приходят в обратном порядке - сначала данные для транзакции 1, а затем для транзакции 0.

Далее идут 4 некорректных запроса:

- Сигнал `arvalid` снимается до прихода `arready`;
- Burst-транзакция пересекает границу в 4к;
- Сигнал `arready` имеет неопределенное значение во время активности `arvalid`;
- Иксы на сигнале `araddr`.

Код для полноценного разбора транзакий на шине получился достаточно объёмный, поэтому в статье его приводить не будем. Просто посмотрим на результат, который показан на рис.10.

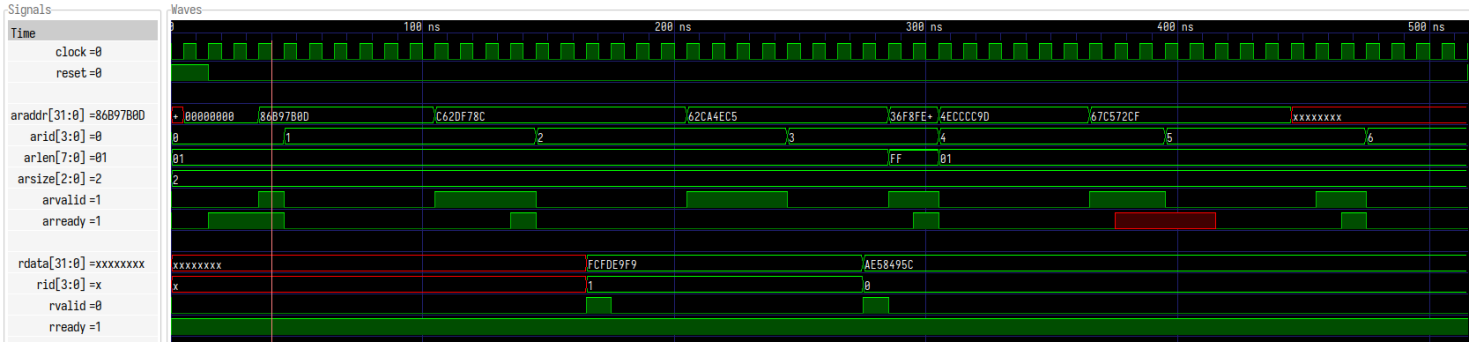


Рис.9 Транзакции на шине AXI

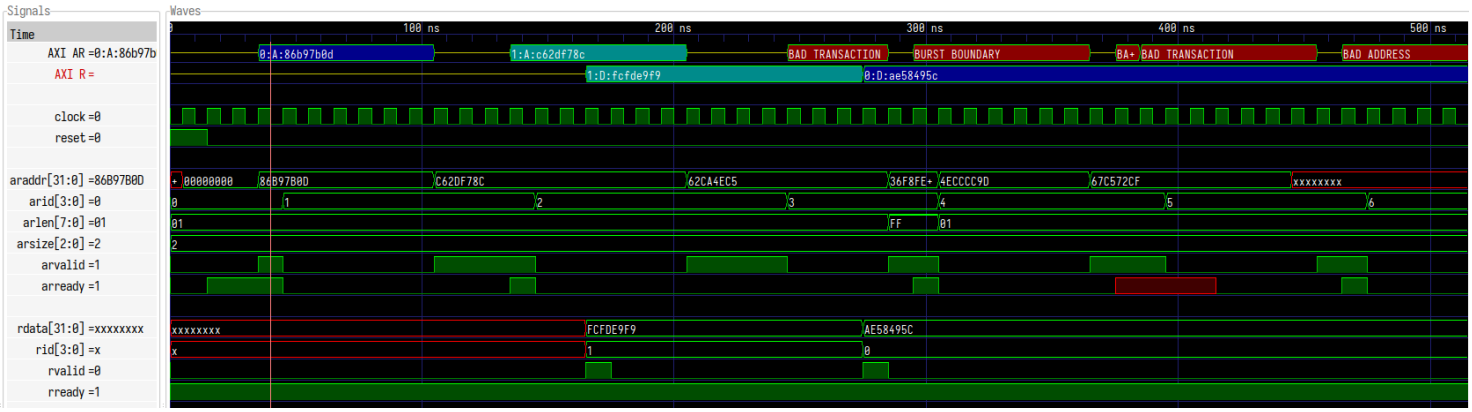


Рис.10 Результат применения фильтра к транзакциям AXI

Как можно увидеть, фильтр вернул два виртуальных сигнала с транзакциями в канале адреса чтения и в канале чтения данных. Неупорядоченные транзакции были обнаружены и корректно раскрашены с соответствии с сигналами `arid` и `rid`. Некорректные транзакции были распознаны и обозначены в соответствии с типом ошибки.

Таким образом, мы получили визуальное представление транзакций, которое может помочь в отладке и поиске ошибок в коде RTL.

Заключение

Фильтры в программе GTKWave представляют достаточно мощные средства для обработки и визуализации временных диаграмм. Простые фильтры трансляции позволяют без труда выполнить замену цифровых значений на диаграмме на понятный человекочитаемый текст. В то же время, для сложной фильтрации с сохранением контекста и с учётом зависимости между сигналами можно использовать транзакционные фильтры. Их сложность компенсируется универсальностью и гибкостью.

Исходные коды вы можете найти на странице проекта на GitHub: https://github.com/punzik/gtkwave-filters-article

set set set; #это не только легально, но и полезно

Коробков Михаил, *admin@fpga-systems.ru*
Telegram @KeisN13

Обсуждение и комментарии :: ссылка

Аннотация

При проектировании на ПЛИС разработчики часто прибегают к написанию проектов на языке управления средой, в качестве которого выступает Tcl. Однако, использование языка Tcl сопряжено с одной большой трудностью – для него не существует встроенных в среду проектирования, например Quartus, Vivado, Libero и др инструментов отладки, в отличие от простого написания скриптов на Tcl для создания различного рода приложений. Эта особенность связана с тем, что не удается подключить библиотеки с процедурами из конкретной среды проектирования на ПЛИС к сторонней EDA, в которой возможность отладки Tcl скриптов присутствует (по крайней мере автору не известны механизмы такой интеграции).

Введение

Зачастую, разработчики прибегают к наиболее очевидному способу отладки – это вывод информационных сообщений в консоль во время запуска Tcl-скрипта через команду puts, которая выводит сообщения в заданный стандартный поток ввода-вывода, которым по умолчанию является консоль или файл.

Написание больших Tcl-скриптов сопряжено со множеством трудностей при их отладке, поэтому в этой заметке мы посмотрим на альтернативные варианты, позволяющие несколько облегчить жизнь FPGA разработчикам при автоматизации проектов с использованием Tcl.

Что такое Tcl?

Tool Command Language – («командный язык инструментов», читается «тикль» или «ти-си-эль») — скриптовый язык высокого уровня [wiki].

Tcl исторически используется для управления средами проектирования и автоматизации сборки проектов. Этот язык является достаточно простым, но архаичным и имеет несколько не привычный синтаксис. Он имеет ряд особенностей, коротко про которые можно узнать из обучающего 20-ти минутного ролика [ссылка]

Удобным инструментом в средах разработки для ПЛИС, например AMD Vivado, является автоматический вывод команды с подставленными значениями аргументов, в консоль сразу по вводу этой команды в консоль. К сожалению, данный механизм не всегда присутствует в среде проектирования некоторых производителей, и для вывода значений назначенных переменной (повторно) применяют команду `puts ${переменная}`.

Трюки отладки Tcl скриптов

Для написания нативных Tcl скриптов, в которые не включены команды из сред проектирования на ПЛИС существует насколько сред разработки. Наиболее популярными являются Komodo от команды ActiveState (см. ActiveState Tcl) и расширение для Eclipse – DLTK. Komodo IDE является мощным редактором с большим количеством инструментов отладки, но, как говорилось выше, прикручивание команд из той же Vivado не представляется возможным и при отладке скриптов, создаваемых для автоматизации проектов с командами Vivado, вызывает определённые трудности. Такое же утверждение верно и для DLTK.

В поиске решений по способам отладки автору не удалось также найти плагинов расширения для популярного редактора VS Code, кроме нескольких статей, в которых была описана возможность использования встроенного отладчика VS Code для создания инструмента отладки Tcl скриптов, но опять же, они не смогут работать со встроенными командами среды проектирования для ПЛИС.

На странице проекта Tcl/Tk автору удалось найти некоторые “хитрые” приёмы для улучшения возможностей отладки Tcl скриптов [ссылка], речь о которых пойдет далее. Практически все они основаны на

подмене/переименовании ключевых слов и команд. И особенность языка Tcl, в которой команда `set set set` является вполне легальной, нам в этом поможет.

Вариант 1

Это вариант наиболее близок к стандартному выводу сообщений с использованием команды `puts`, однако здесь добавляется функционал, который позволяет включить или отключить вывод сообщений в консоль.

```
proc dputs {msg} {
    global debugMode
    if { $debugMode } { puts $msg }
}
```

Для его использования достаточно слать переменную с именем `debugMode` и присвоить ей значение 0 для запрета вывода отладочных сообщений и 1 для включения. Не забудьте, что процедура должна быть доступна перед вызовом.

Пример использования

```
set debugMode 1

proc dputs {msg} {
    global debugMode
     if { $debugMode } { puts $msg }
}

set a "Some string"
dputs "Display debug information"

В консоли появится
Display debug information
```

```
set debugMode 0

proc dputs {msg} {
    global debugMode
    if { $debugMode } { puts $msg }
}

set a "Some string"
dputs "Display debug information"

В консоли ни чего не появится
```

Данную процедуру просто расширить до функционала, когда вся отладочная информация будет отправляться не в консоль, а в файл, например

```
set debugMode 1
set file_path {путь к файлу для сохранения информации}
set fp [open "$file_path" w]; #создаем или очищаем файл
close $fp

proc dputs {msg} {
    global debugMode
    if { $debugMode } {
        set fp [open "$file_path" w]
        puts $msg
        close $fp
    }
}
```

Вариант 2

Этот прием использует подмену стандартной команды `set`. При этом появляется возможность дополнять команду необходимым функционалом

```
rename set _set
proc set {var args} {
    puts [list set $var $args]
    uplevel _set $var $args
} ;
```

Следует помнить, что такое действие подменяет все вызовы команды set используемые в скриптах, включая ее вызов в процедурах. Использование такого подхода может повлечь вывод большого количества лишней информации.

Новая процедура не только устанавливает значение переменной, но и выводит имя переменной и установленное значение

Вернуть обычное поведение команды set можно использованием следующей процедуры

```
proc set {var args} {
    uplevel _set $var $args
} ;
```

Вариант 3

Начиная с Tcl 8.4 появилась команда trace, которая отслеживает доступ к переменным, использование команд и их выполнение. Эту команду можно использовать с подходом, применяемым в вышеописанном варианте с командой `set`, но применительно к вызову процедур.

Команда `trace` имеет богатый функционал, который используется для вывода отладочной информации, а в приведенном ниже примере используется команда `trace add execution name ops commandPrefix`, которая организует выполнение `commandPrefix` (с дополнительными аргументами) всякий раз, когда выполняется команда, с трассировкой, происходящей в точках, указанных в списке операций. Если команда не существует, будет выдано сообщение об ошибке.

```
rename proc _proc
_proc proc {name arglist body} {
    uplevel 1 [list _proc $name $arglist $body]
    uplevel 1 [list trace add execution $name enterstep
[list ::proc_start $name]]
}
_proc proc_start {name command op} {
    puts "$name >> $command"
}
```

В процедуре `proc_start` задается форматирование вывода вызванной команды, который без особого труда может быть изменён самостоятельно под ваши задачи.

Пример работы приведен ниже

```
rename proc _proc
_proc proc {name arglist body} {
    uplevel 1 [list _proc $name $arglist $body]
    uplevel 1 [list trace add execution $name enterstep
[list ::proc_start $name]]
}
_proc proc_start {name command op} {
    puts "$name >> $command"
}

proc add {f d} {
  set e "[expr $f + $d]"
}

set f [add 2 3]

set a 8
set b 9
puts $a
puts $b
```

В консоли будет следующий вывод

```
add >> expr 2 + 3
add >> set e 5
8
9
```

Данный подход позволяет не только выводить отладочную информацию, но и просматривать последовательность подстановок и последовательность выполнения команд.

Заключение

В этой короткой заметке приведены три способа, которые позволят вам улучшуть отладку самописных Tcl скриптов. Осталось только интегрировать их. Если вам известны другие способы отладки Tcl, применяемых при разработке скриптов автоматизации сред проетирования для FPGA, напишите об этом в комментариях, а еще лучше подготовьте небольшую заметку.

Verilator – многофункциональный инструмент эмуляции и тестирования Verilog-кода

Кашканов Артём
Телеграм: @radiolok

Обсуждение и комментарии :: ссылка

Аннотация

В статье рассматривается открытый программный пакет Verilator, который позволяет как провести статический анализ Verilog-кода, так и реализовать систему тестирования и верификации создаваемых модулей.

Ключевые слова: Verilator, ASIC, FPGA, Iverilog, GTKWave

Введение

Создание «Интегральной схемы для конкретного применения», или ASIC – влечет за собой *некоторые* трудности касаемо доступа к необходимому инструментарию. Высокая стоимость лицензий таких проприетарных продуктов как Synopsys Design Compiler или Mentor Graphics, а также различные экспортные ограничения по их использованию, равно как и высокая сложность вхождения в область аппаратной разработки, приводят к очень медленному и болезненному росту сообщества разработчиков цифровых схем для FPGA или ASIC. В то же время разработчику программного обеспечения предлагается бесчисленное множество открытых и доступных средств разработки, компиляции, развертывания, анализа производительности и т.п. на практически любом существующем языке программирования.

В последние годы отмечается бурное развитие разработки открытых инструментов для проектирования микросхем. Не смотря на невозможность в столь сжатые сроки предложить функциональность, аналогичную коммерческим комбайнам с полувековой историей, такие проекты как Iverilog[1], Verilator[2], Yosys[3], OSS CAD Suite[4], OpenROAD[5] и др. уже позволяют существенно упростить процесс вхождения в разработку цифровой аппаратуры, покрывая весь спектр необходимых для этого задач.

Техническое задание

Допустим, необходимо создать сценический аудио-эффект на базе отладочной платы с FPGA, оснащенной 16-битными АЦП и ЦАП. Написанный код необходимо покрыть автоматическими тестами, при этом неплохо бы иметь возможность проводить тесты на реальных данных, а также время от времени производить прослушивание получившихся в результате экспериментов звуковых эффектов.

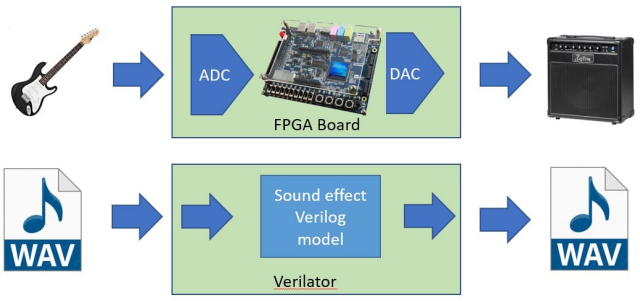


Рисунок 1 Модель передачи данных с физической ПЛИС(сверху) и ее виртуальной моделью (снизу)

Напишем модуль, который будет по сигналу тактирования считывать данные во внутренний регистр, а еще через цикл – выдавать измененные данные обратно. Для простоты эксперимента поставим перемычку со входа на выход:

```
module distortion #(
    parameter WIDTH=16//INT16 size
)(
    input wire rst_n,
    input wire clk,
    input wire [WIDTH-1:0] IN,
    output reg [WIDTH-1:0] OUT
);
reg [WIDTH-1:0] in;
always @(posedge clk) begin
    in <= (!rst_n) ? {(WIDTH){1'b0}}: IN;
end
wire [WIDTH-1:0] out = in; //In => Out
always @(posedge clk) begin
    OUT <= (!rst_n) ? {(WIDTH){1'b0}}: out;
end
endmodule
```

Разумеется, вместо перемычки в реальности будет реализован какой-либо из аудио-эффектов, например: эхо, реверберация, хорус, овердрайв, фильтрация и др. При этом модуль поддерживает конвейризацию и выдает результат по текущему сэмплу через цикл.

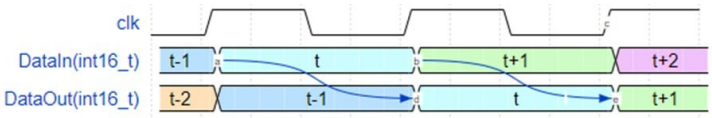


Рисунок 2 Конвейеризация данных

Симуляция с Icarus Verilog

Самый простой способ понять что модуль работает, оставшись при этом в Verilog-окружении, это написать тестбенч-файл, который будет подавать случайные данные на вход тестируемого модуля. Справиться с этим поможет Icarus Verilog – открытый симулятор для языка Verilog и SystemVerilog. Iverilog заявляет о поддержке стандарта IEEE-1364 «насколько это вообще возможно», а также некоторых расширений языка System Verilog. Поддерживаются .SDF файлы для симуляции процесса распространения сигнала. Имеется поддержка генерации .VCD файлов. Главный недостаток инструмента – скорость его работы на порядки ниже коммерческих симуляторов, например таких как VCS/ Modelsim и т.п.

```
module distortion_tb();
parameter WIDTH=16;
reg rst_n;
reg clk;
initial begin
    clk = 1'b1;
    forever #1 clk = ~clk;
end

initial begin
    rst_n = 1'b0;
    #5 rst_n = 1'b1;
    #1000 $finish;
end

reg [WIDTH-1:0] IN;
wire [WIDTH-1:0] OUT;
distortion   #(.WIDTH(WIDTH))   dut(.rst_n(rst_n),.clk
(clk),.IN(IN),.OUT(OUT));

initial begin $dumpfile("distortion_tb.vcd");
$dumpvars(0, distortion_tb); end

always @(posedge clk) begin
    IN <= (!rst_n)? 0 : $random();
      //Важно:  random()не  поддерживает  разрядность
больше 32 бит
end
endmodule
```

Компиляция кода и запуск симуляции осуществляются следующим образом:

```
iverilog -o distortion -s distortion_tb distortion.v
distortion_tb.v -g2012 ./distortion
```

В качестве программы для просмотра сгенерированных .VCD файлов можно использовать GTKWave – легковесный инструмент визуализации временных диаграмм состояний. Программа имеет интуитивно-понятный интерфейс, прекрасно работает как в Linux так и Windows. Скорость работы практически не зависит от размера файлов – автор без проблем открывал дампы по 50-100GiB

На рисунке 3 видно, как после снятия сигнала сброса и подачи данных на вход IN, сигнал на выходе появляется с задержкой в два цикла. Как минимум, модуль корректно пропускает через себя данные.

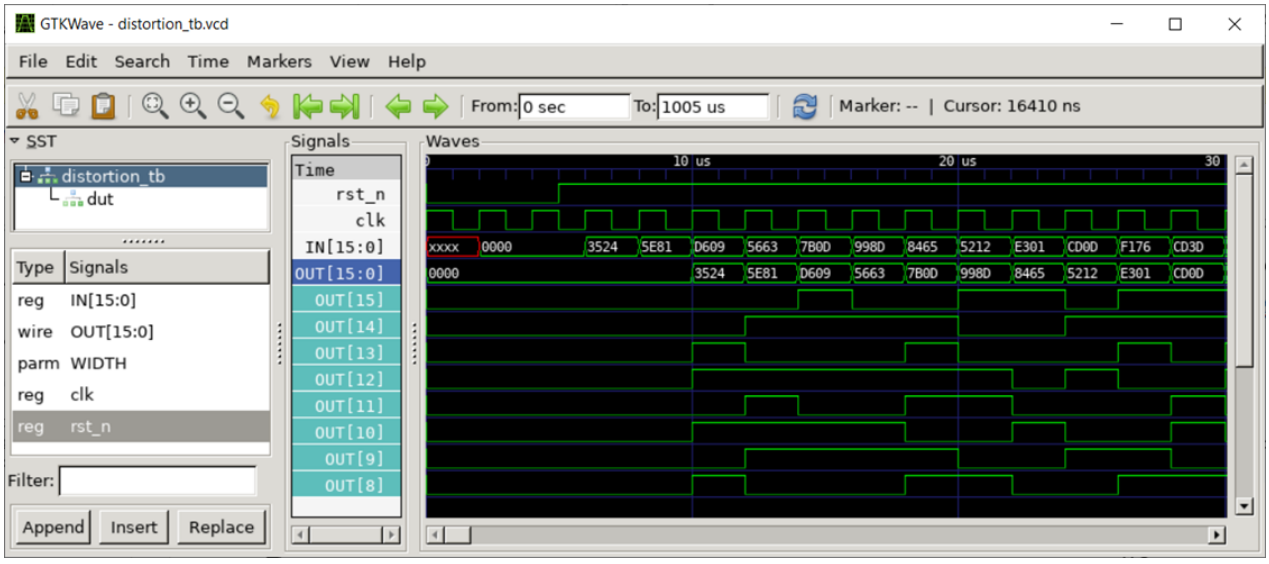


Рисунок 3 Главное окно программы GTKWave

Разумеется, случайный набор данных не позволяет полноценно протестировать устройство, поэтому необходимо подавать на вход модуля настоящие сэмплы звуковых файлов, например, разобрав с помощью любой подходящей библиотеки [6] WAV-файл преобразовать его в удобоваримый текстовый формат для тестбенча. Впоследствии, с помощью $readmemh() или $fscanf () - грузить эти данные в модуль.

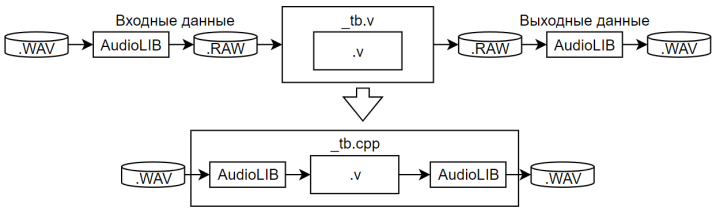


Рисунок 4 Действительная (сверху) и желаемая (снизу) схема работы с аудиофайлами

Глядя на получившуюся схему (рисунок 4, сверху), возникает вопрос – а нельзя ли избавиться от множества промежуточных шагов?

Verilator

Verilator это преобразователь Verilog или SystemVerilog кода в файлы C++ или SystemC, со встроенными функциями статического анализа (линтинга) и покрытия кода. «Верилированные» файлы далее компилируются с помощью любого C++ (gcc/clang/MSVC++) компилятора в библиотеку для встраивания в систему тестирования или в самостоятельное приложение.

Сразу стоит отметить, что Verilator следует стандарту IEEE-1364 лишь частично. Он не поддерживает SDF-аннотацию, смешанные сигналы, все изменения состояний происходят одномоментно по синхроимпульсу. На вход он принимает только синтезируемый код. С другой стороны такой подход дает огромную скорость работы модели – на порядки выше транслируемого Icarus Verilog. Дополнительное ускорение достигается благодаря поддержке многопоточности, что позволяет работать с действительно большими проектами.

Настройка окружения

Наиболее простой способ установки Verilator – из официального репозитория Ubuntu. К сожалению, даже в Ubuntu 23.10 для установки предлагается довольно старая версия пакета, в которой может не оказаться линтера. Поэтому более правильным вариантом будет установка последнего доступного релиза из GitHub-репозитория по представленной там инструкции:

```
sudo apt-get install git perl python3 gperf make autoconf g++ flex bison ccache
sudo apt-get install python-is-python3 python3-pip
sudo apt-get install libgoogle-perftools-dev numactl perl-doc help2man
sudo apt-get install libfl2  # Ubuntu only (ignore if gives error)
sudo apt-get install libfl-dev  # Ubuntu only (ignore if gives error)
sudo apt-get install zlibc zlib1g zlib1g-dev  # Ubuntu only (ignore if gives error)

git clone https://github.com/verilator/verilator   # Only first time

# Every time you need to build:
unsetenv VERILATOR_ROOT  # For csh; ignore error if on bash
unset VERILATOR_ROOT  # For bash
cd verilator
git pull         # Make sure git repository is up-to-date
latest=$(git tag --sort=taggerdate | tail -1) # Get latest release tag
git checkout ${latest}  # Switch to specified release version

autoconf         # Create ./configure script
./configure      # Configure and create Makefile
make -j `nproc`  # Build Verilator itself (if error, try just 'make')
make test
sudo make install
```

Аналогичные претензии можно предъявить пакетам симулятора Iverilog и синтезатору Yosys – также рекомендуется ставить самые свежие версии вручную из репозиториев. Отчасти это связано с активным развитием всех продуктов и оперативным закрытием багов.

```
#include <stdlib.h>
#include <iostream>
#include <stdio.h>
#include <ctype.h>
#include <cmath>
#include <AudioFile.h>
#include <verilated.h>
#include "verilated_vpi.h"
#include <verilated_vcd_c.h>
#include "Vdistortion.h"//заголовочный файл с описанием интерфейса модуля

vluint64_t sim_time = 0;
#define PIPELINE (2)
int main(int argc, char** argv, char** env) {
    AudioFile<float> a;
    a.load ("input.wav");
    Verilated::traceEverOn(true);
    Vdistortion* dut = new Vdistortion();
    VerilatedVcdC *m_trace = new VerilatedVcdC;
    dut->trace(m_trace, 5);
    m_trace->open("vdistortion.vcd");
    for (int channel = 0; channel < a.getNumChannels(); channel++){
        int samples =a.getNumSamplesPerChannel();
        dut->clk = 0;
        dut->rst_n = 0;
        for (int i = 0; i < samples + PIPELINE; i++){
            if (i < samples){
                float sf = a.samples[channel][i];//WAV сигнал от -1 to +1
                sf = sf * 32767;//Простая конвертация в short от -32767 to +32767
                dut->IN = static_cast<int16_t>(sf);//отправили
            }
            if (i >= PIPELINE){
                float sf = static_cast<float>(dut->OUT);//забрали
                a.samples[channel][i-PIPELINE] = sf / 32767;
            }
            if (sim_time == 4) {dut->rst_n = 1;}
            dut->clk = 1;
            dut->eval();//запустили просчет цикла
            m_trace->dump(sim_time);//сохранили текущее состояние
            sim_time++;
            dut->clk = 0;
            dut->eval();
            m_trace->dump(sim_time);
            sim_time++;
        }
    }
    m_trace->close();
    delete dut;
    a.save ("output.waav");
    exit(EXIT_SUCCESS);
}
```

Единственное что можно поставить из репозитория - так это утилиту для просмотра vcd-файлов GTKWave

```
sudo apt-get install gtkwave
```

Для запуска симуляции с помощью Verilator потребуется создание нового тестбенча. Помимо написания проекта на родном языке C++ , реализовать тестирование можно и с помощью языка SystemC, но его использование выходит за рамки данной статьи. Адепты Python могут воспользоваться поддержкой Verilator в фреймворке cocotb. Для этого достаточно иметь в системе Verilator >= 5.006 и при сборке проекта указать `make SIM=verilator`

Напишем файл на языке C++ который будет в рантайме создавать объект тестируемого Verilog-модуля и с помощью внешней библиотеки <AudioFile.h> читать данные из входного .WAV файла, отправлять сэмплы в модуль, и забирать их, упаковывая в выходной .WAV файл. Так как в аудиофайле данные лежат в формате FP16 и могут принимать значения до -1.0 до +1.0, а физическое устройство будет работать с 16-битными знаковыми целочисленными значениями в диапазоне от -32768 до +32767, произведем простейшую конвертацию одного в другое с помощью масштабирования:

Компиляция проекта осуществляется силами самого Verilator с помощью следующего скрипта:

```
verilator -Wall --trace --top distortion --cc distortion.v \
    --timescale 1us/1ns \
    --exe distortion.cpp -CFLAGS -I$AudioFile/
make -j`nproc` -C obj_dir -f Vdistortion.mk Vdistortion
./obj_dir/Vdistortion
```

После завершения работы программы в папке назначения появится output.wav файл с итоговым содержимым, которое в случае с перемычной должно быть неотличимым на слух от входного файла. Предполагается, что в случае реализации нормального аудио-эффекта, итоговый результат можно прослушать, сравнить с образцом, или, например провести частотный анализ и определить качество работы разрабатываемого фильтра. Скорость симуляции напрямую зависит от сложности Verilog-модуля, однако обработка аудио, как в данном примере, может производиться в прямом эфире даже для довольно сложных эффектов. Исходные коды проекта, примеры фильтров,та также Docker-скрипты для развертывания необходимого окружения представлены в репозитории [7].

Заключение

В статье рассмотрен простой пример использования Verilator в качестве симулятора Verilog-модулей. Помимо автоматизации тестирования силами более высокоуровневых языков программирования, таких как C++ или Python, у данного инструмента есть еще одно важное свойство – возможность являться виртуальной машиной для запуска и тестирования сложных систем. Например, в [8] в качестве Verilog-модуля используется описание вновь разрабатываемого процессора оригинальной микроархитектуры. На C++ написаны оболочка для эмуляции управляющих сигналов процессора и сигналов шины данных, адреса; шины ввода/вывода. Также, создан псевдографический интерфейс на фреймворке ncurses. Такое решение позволяет производить разработку процессора более комплексно, без необходимости запускаться на реальном оборудовании. С исчезновением зависимости от отладочной платы процесс компиляции и запуска новой версии кода занимает единицы секунд вместо десятков минут полного процесса синтеза и трассировки файлов для записи в ПЛИС, что существенно повышает производительность труда. Чем большее число экспериментов может быть поставлено в единицу времени – тем более эффективно может производиться разработка цифровой схемы.

Список литературы

1. https://github.com/steveicarus/iverilog
2. https://www.veripool.org/verilator/
3. https://github.com/YosysHQ/yosys
3. https://github.com/YosysHQ/oss-cad-suite-build
5. https://github.com/The-OpenROAD-Project
6. https://github.com/adamstark/AudioFile
7. https://github.com/radiolok/distortionhdl
8. https://github.com/radiolok/dekatronpc

Язык SystemRDL в разработке ip-блоков

Высоцкий Матвей
Телеграм: @raconteur_visotsky

Обсуждение и комментарии :: ссылка

Проблема

В наши дни разработчик электронной аппаратуры сталкивается с множеством проблем, которые требуют значительных временных затрат на реализацию идеи и последующие отладку и тестирование. Одна из таких проблем - следование принципам стандартизации ip на уровне шинных интерфейсов. Разработчику не достаточно разработать функциональную часть, важно создать коммутационную среду, которая позволяла бы использовать разработанное устройство в широком спектре задач и возможных комбинаций с другими элементами системы. Коммутационная среда организуется реализацией требуемых шинных интерфейсов и заданием алгоритма квитирования данных. Сейчас большой популярностью пользуются интерфейсы, разработанные компанией ARM. Примерами таких шин являются APB3, APB4, AXI3, AXI4, AXI4-lite, AHB4, AHB5 и т.д. Каждый интерфейс имеет свои плюсы и минусы, каждый имеет свою документацию для последующей реализации. Некоторые интерфейсы, такие как APB3 могут быть легко реализованы и быстро внедрены в проект, другие, например, AXI4 требуют подробного изучения объемной документации, долгой HDL-имплементации и больших временных затрат на верификацию. Описанные условия ограничивают разработчика в процессе разработки: в целях оптимизации временных затрат реализуется 1 шинный интерфейс(в редких случаях 2). Также высокая сложность имплементации оказывает влияние на гибкость последующего сопровождения проекта: сложные высокоскоростные интерфейсы не меняются, а соединяются с переходными мостами, преобразующими один шинный интерфейс в другой, что сказывается на пропускной способности интерфейса. Таким образом, сложность заключается в создании обязательной для интеграции части, логика работы самого устройства при этом не затрагивается.

Решение проблемы

Решение было предложено объединением SPIRIT Consortium [1]: компании, входящие в объединение, разработали язык описания регистров SystemRDL [2], который обладает лаконичным синтаксисом и строгой иерархичностью в описании адресного пространства. Также компилятор языка автоматически генерирует необходимый шинный интерфейс, что решает проблему изучения и имплементации протоколов обмена данными, позволяя разработчику ограничиться только пониманием осуществляемых транзакций. Кроме того, реализация готового ip происходит быстрее, а коммутационная среда вариативнее: появляется возможность перераспределить временные ресурсы с большим уклоном в сторону функциональной части ip, и создавать любой шинный интерфейс необходимый в каждом конкретном случае.

Компании использующие SystemRDL

Основными достоинствами SystemRDL являются лаконичность синтаксиса и четкая иерархичность, что сделало язык популярным: он начал широко применяться различными компаниями, занимающимися разработкой ip-блоков и САПР для микроэлектроники. На данный момент в своих разработках язык используют: ARM, Cadence, Mentor Graphics, STM, Synopsys, Texas Instruments, Nvidia, AMD и другие. В России SystemRDL еще не распространен и только набирает популярность.

Примеры использования языка

Для наглядности простоты использования языка, были созданы примеры описания небольших адресных пространств.

```
1.  addrmap Addrmap{
2.      name = "Addrmap";
3.      desc = "Addrmap";
4.
5.      default regwidth = 8;
6.      default sw = rw;
7.      default hw = r;
8.
9.      reg {
10.         name = "Register name";
11.         desc = "Register description";
12.
13.         field{
14.             name = "Field name";
15.             desc = "Field description";
16.         }fieldname[1:0] = 0x0;
17.     }My_reg @ 0x0;
18. };
```

Листинг 1. Пример создания 8-битного регистра.

Приведенный участок кода демонстрирует нам описание адресного пространства, состоящего из одного 8-битный регистра с одним значащим полем из 2 бит. В строке 1 создается адресное пространство с именем Addrmap. В строках 2 и 3 дополнительно задается полное имя (если оно есть) и описание адресного пространства. Эта информация может быть использована в документации. В строках 5-7 определяются постоянные значения (их всегда можно переопределить локально). В строке 5 задается ширина регистра, в строках 6-7 определяется программный и аппаратный доступ по умолчанию. В данном случае по умолчанию считается, что программно все поля доступны для чтения и записи, аппаратно - только для чтения. В строках 9-17 создается неявное описание регистра с именем My_reg и адресом 0x0. В строках 10-11 содержится имя и описание регистра. Строки 13-17 демонстрируют неявное создание поля внутри регистра. В строках 14-15 находится описание поля. В строке 17 находится обозначение поля, его размеры и значение сброса.

Результат генерации доступен в приложении, часть A.

Для описания данного адресного пространства достаточно 19 строк на языке SystemRDL. Результат генерации содержит 174 строки, выполненные на синтезируемом подмножестве языка SystemVerilog. HDL код включает в себя описание одного 8-битного регистра, шинного интерфейса APB3 и алгоритма квитирования данных. Описание работы алгоритма квитирования и шинного интерфейса трудно для восприятия человеком, описание на SystemRDL лучше структурировано и содержит меньше элементов, которые требуют детального рассмотрения. Таким образом, язык SystemRDL облегчает работу разработчика электронной аппаратуры, позволяя сконцентрироваться на функциональной части устройства.

```
1.  addrmap Addrmap {
2.      name = "Addrmap";
3.      desc = "My addrmap";
4.      default regwidth = 32;
5.      default sw = rw;
6.
7.      reg ISR {
8.          name = "ISR";
9.          desc = "Interrupt and status register";
10.         default hw = w;
11.         default onwrite = woclr;
12.         default precedence = sw;
13.
14.         field {
15.             name = "Interrupt";
16.             desc = "Interrupt field";
17.             posedge intr;
18.         } INTRR[0:0] = 0x0;
19.     };
20.
21.     reg IER {
22.         name = "IER";
23.         desc = "Interrupt enable register";
24.         default hw = na;
25.
26.         field {
27.             name = "Interrupt enable";
28.             desc = "Interrupt enable field";
29.         } INTRRIE[0:0] = 0x0;
30.     };
31.
32.     IER Reg_IER @0x4;
33.     ISR Reg_ISR @0x0;
34.
35.     Reg_ISR.INTRR->enable = Reg_IER.INTRRIE;
36. };
```

Листинг 2. Пример создания регистра статуса и прерывания с очисткой по записи единицы и регистра разрешения прерываний.

Данный пример показывает реализацию регистра статуса и прерываний и регистра разрешения прерываний. Аналогично примеру 1 создается адресное пространство и задаются настройки по умолчанию, относящиеся ко всему пространству. В строках 7-19 демонстрируется явное описание статусного регистра. Такой способ описания позволяет создавать дубликаты регистров/полей/сигналов. В строке 10 задается настройка по умолчанию для аппаратного доступа, которое будет действовать внутри этого регистра. В строке 11 задается способ сброса поля - сброс при записи в поле 1. В строке 12 определяется приоритет программного доступа к полям. Строки 14-18

демонстрируют неявное задание однобитного поля прерывания. В строке 17 определяется способ включения прерывания: прерывание по фронту. В строках 21-30 происходит явное задание регистра разрешения прерывания. Регистр настраивается аналогично примеру 1, за исключением аппаратного доступа в поле. Аппаратный доступ, в данном случае, полностью запрещен. В строках 32-33 создаются экземпляры объявленных регистров и задаются адресные смещения. В строке 35 связывается поле прерывания регистра разрешения прерываний с сигналом разрешения прерывания в регистре статуса.

В целях демонстрации различных возможностей языка SystemRDL данный код был преобразован в UVM класс, который доступен для просмотра в приложении, часть Б.

Генерация тестовых структур полезна верификаторам, так как может служить заготовкой для тестового покрытия. Разработчику данная опция позволяет оценить работоспособность функционального блока, избегая ручного тестирования.

Приведенные примеры показывают наличие четкой структуры и иерархичности полей и регистров. Такое устройство описания адресного пространства позволяет легко изменять структуру адресного пространства и регистров и изменять настройки полей. Описание регистров очень лаконично, так как многие технические детали реализации скрыты, благодаря чему код легко воспринимается человеком.

Инструменты для работы с языком

Первая версия спецификации языка вышла в 2013 году. За 10 лет было разработано несколько коммерческих и несколько OpenSource инструментов для работы с языком. Коммерческие инструменты: Agnisys, Semifore's CSR Compiler и Magillem. Среди open-source можно выделить ORDT(Open Register Design Tool) и PeakRDL complier.

ORDT

- Open Register Design Tool[3] – OpenSource генератор языка SystemRDL, написанный на языке Java, что позволяет использовать его на любой платформе. Генератор может принимать на вход описание регистров, выполненное на языке SystemRDL по стандарту Accelera[4] и JSpec(формат описания регистров, используемый в JuniperNetworks). Полученное описание конвертируется в: SystemVerilog/Verilog описание регистров
- UVM классы (тестовая модель регистров)
- Заголовочный файл на языке C, содержащий описание регистров для программирования
- Текстовые и XML файлы описания регистров

PeakRDL

PeakRDL[5] - кроссплатформенный OpenSource генератор языка SystemRDL написанный на языке Python и входящий в стандартную библиотеку языка Python. Генератор может принимать на вход файлы описания регистров на языке SystemRDL и файлы моделей регистров в формате XML. Полученные файлы конвертируются в:

- Описание регистров на синтезируемом SystemVerilog
- HTML документация (сайт написанный с использованием HTML, CSS, JS)
- UVM модель регистров
- Заголовочный файл на языке C для программирования регистров
- Конвертация описания на SystemRDL в XML модель
- Конвертация XML модели в описание на SystemRDL

PeakRDL - гибкий генератор, который может быть расширен. Пользователь может добавить новые виды генерации или добавить свои шинные интерфейсы. На сайте компилятора есть много обучающих материалов и примеров, которые могут быть полезны начинающим пользователям.

Недостатки

Недостатки SystemRDL можно разделить на 2 основные категории: недостатки инструментов разработки и недостатки функционала языка.

Недостатки связанные с инструментами разработки

Возможность расширения генератора имеет свои достоинства и недостатки. Пользователь может расширить список шинных интерфейсов, но расширение принимается генератором, как по умолчанию правильное и верифицированное. В случае ошибки при создании расширения программа не сообщит об этом пользователю и сгенерирует HDL - описание в соответствии указанным в расширении правилам.

Недостатки функциональности языка SystemRDL

При работе с данным языком разработчик должен понимать, что SystemRDL реализует узкий набор возможностей и не может заменить собой классические языки описания аппаратуры. Цель данного языка - упростить разработку адресного пространства и шинного интерфейса ip. SystemRDL не может использоваться для создания сложных функциональных частей устройства, поэтому для разработки логики работы используются HDL языки.

Вывод

Язык описания регистров SystemRDL это удобный инструмент для решения конкретных задач в процессе разработки ip. Использование SystemRDL позволяет избавиться от трудностей в проектировании программируемой части ip блоков и создании коммутационной среды, что существенно сокращает временные издержки на разработку и тестирование. Также SystemRDL позволяет упростить последующее сопровождение разработанного устройства.

Источники

[1] https://en.wikipedia.org/wiki/Accellera#The_SPIRIT_Consortium Статья сдержащая информацию о SPIRIT Consortium.

[2] https://en.wikipedia.org/wiki/SystemRDL Определение и краткая история языка SystemRDL.

[3] https://github.com/Juniper/open-register-design-tool Репозиторий компании Juniper Networks, включающий исходные файлы и документацию конвертера ORDT.

[4] https://accellerasystemsinitiative.org/downloads/standards/systemrdl Документация языка SystemRDL.

[5] https://peakrdl.readthedocs.io/en/latest/ PeakRDL конвертер, официальный сайт-документация.

Приложение

Часть А.

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


    input wire s_apb_psel,
    input wire s_apb_penable,
    input wire s_apb_pwrite,
    input wire [0:0] 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,


    output Addrmap_pkg::Addrmap__out_t hwif_out
  );


  //--------------------------------------------------------
--------------------
  // CPU Bus interface logic
  //--------------------------------------------------------
--------------------
  logic cpuif_req;
  logic cpuif_req_is_wr;
  logic [0:0] cpuif_addr;
  logic [7:0] cpuif_wr_data;
  logic [7:0] cpuif_wr_biten;
  logic cpuif_req_stall_wr;
  logic cpuif_req_stall_rd;


  logic cpuif_rd_ack;
  logic cpuif_rd_err;
  logic [7:0] cpuif_rd_data;


  logic cpuif_wr_ack;
  logic cpuif_wr_err;


  // Request
  logic is_active;
  always_ff @(posedge clk) begin
    if(rst) begin
      is_active <= '0;
      cpuif_req <= '0;
      cpuif_req_is_wr <= '0;
      cpuif_addr <= '0;
      cpuif_wr_data <= '0;
    end else begin
      if(~is_active) begin
        if(s_apb_psel) begin
          is_active <= '1;
          cpuif_req <= '1;
          cpuif_req_is_wr <= s_apb_pwrite;
          cpuif_addr <= s_apb_paddr[0:0];
          cpuif_wr_data <= s_apb_pwdata;
        end
      end else begin
        cpuif_req <= '0;
        if(cpuif_rd_ack || cpuif_wr_ack) begin
```

```
        is_active <= '0;
      end
    end
  end
  assign cpuif_wr_biten = '1;


  // Response
  assign s_apb_pready = cpuif_rd_ack | cpuif_wr_ack;
  assign s_apb_prdata = cpuif_rd_data;
  assign s_apb_pslverr = cpuif_rd_err | cpuif_wr_err;


  logic cpuif_req_masked;


  // Read & write latencies are balanced. Stalls not re-
quired
  assign cpuif_req_stall_rd = '0;
  assign cpuif_req_stall_wr = '0;
  assign cpuif_req_masked = cpuif_req
              & !(!cpuif_req_is_wr & cpuif_req_stall_rd)
              & !(cpuif_req_is_wr & cpuif_req_stall_wr);


  //--------------------------------------------------------
--------------------
  // Address Decode
  //--------------------------------------------------------
--------------------
  typedef struct {
    logic My_reg;
  } decoded_reg_strb_t;
  decoded_reg_strb_t decoded_reg_strb;
  logic decoded_req;
  logic decoded_req_is_wr;
  logic [7:0] decoded_wr_data;
  logic [7:0] decoded_wr_biten;


  always_comb begin
    decoded_reg_strb.My_reg = cpuif_req_masked &
(cpuif_addr == 'h0);
  end


  // Pass down signals to next stage
  assign decoded_req = cpuif_req_masked;
  assign decoded_req_is_wr = cpuif_req_is_wr;
  assign decoded_wr_data = cpuif_wr_data;
  assign decoded_wr_biten = cpuif_wr_biten;


  //--------------------------------------------------------
--------------------
  // Field logic
  //--------------------------------------------------------
--------------------
  typedef struct {
    struct {
      struct {
        logic [1:0] next;
        logic load_next;
      } fieldname;
    } My_reg;
  } field_combo_t;
  field_combo_t field_combo;


  typedef struct {
    struct {
      struct {
        logic [1:0] value;
      } fieldname;
    } My_reg;
  } field_storage_t;
  field_storage_t field_storage;


  // Field: Addrmap.My_reg.fieldname
  always_comb begin
    automatic logic [1:0] next_c =
field_storage.My_reg.fieldname.value;
    automatic logic load_next_c = '0;
    if(decoded_reg_strb.My_reg && decoded_req_is_wr)
begin // SW write
      next_c = (field_storage.My_reg.fieldname.value &
~decoded_wr_biten[1:0]) | (decoded_wr_data[1:0] & decod-
ed_wr_biten[1:0]);
      load_next_c = '1;
    end
    field_combo.My_reg.fieldname.next = next_c;
    field_combo.My_reg.fieldname.load_next = load_next_c;
  end
  always_ff @(posedge clk) begin
    if(rst) begin
      field_storage.My_reg.fieldname.value <= 'h0;
    end else if(field_combo.My_reg.fieldname.load_next)
begin
      field_storage.My_reg.fieldname.value <=
field_combo.My_reg.fieldname.next;
    end
  end
  assign hwif_out.My_reg.fieldname.value =
field_storage.My_reg.fieldname.value;


  //--------------------------------------------------------
--------------------
  // Write response
  //--------------------------------------------------------
--------------------
  assign cpuif_wr_ack = decoded_req & decoded_req_is_wr;
  // Writes are always granted with no error response
  assign cpuif_wr_err = '0;


  //--------------------------------------------------------
--------------------
  // Readback
  //--------------------------------------------------------
--------------------


  logic readback_err;
  logic readback_done;
  logic [7:0] readback_data;

  // Assign readback values to a flattened array
  logic [7:0] readback_array[1];
  assign readback_array[0][1:0] = (decoded_reg_strb.My_reg
&& !decoded_req_is_wr) ?
field_storage.My_reg.fieldname.value : '0;
```

```
  assign readback_array[0][7:2] = '0;


  // Reduce the array
  always_comb begin
    automatic logic [7:0] readback_data_var;
    readback_done = decoded_req & ~decoded_req_is_wr;
    readback_err = '0;
    readback_data_var = '0;
    for(int i=0; i<1; i++) readback_data_var |= read-
back_array[i];
    readback_data = readback_data_var;
  end


  assign cpuif_rd_ack = readback_done;
  assign cpuif_rd_data = readback_data;
  assign cpuif_rd_err = readback_err;
Endmodule
```

Часть Б

```
package example_uvm_pkg;
  `include "uvm_macros.svh"
  import uvm_pkg::*;

  // Reg - Addrmap::ISR_INTRR_83853da9
  class Addrmap__ISR_INTRR_83853da9 extends uvm_reg;
    rand uvm_reg_field INTRR;


    function          new(string          name          =
"Addrmap__ISR_INTRR_83853da9");
      super.new(name, 32, UVM_NO_COVERAGE);
    endfunction : new


    virtual function void build();
      this.INTRR = new("INTRR");
      this.INTRR.configure(this, 1, 0, "W1C", 1, 'h0, 1,
1, 0);
    endfunction : build
  endclass : Addrmap__ISR_INTRR_83853da9


  // Reg - Addrmap::IER
  class Addrmap__IER extends uvm_reg;
    rand uvm_reg_field INTRRIE;


    function new(string name = "Addrmap__IER");
      super.new(name, 32, UVM_NO_COVERAGE);
    endfunction : new


    virtual function void build();
      this.INTRRIE = new("INTRRIE");
      this.INTRRIE.configure(this, 1, 0, "RW", 0, 'h0, 1,
1, 0);
    endfunction : build
  endclass : Addrmap__IER


  // Addrmap - Addrmap
  class Addrmap extends uvm_reg_block;
    rand Addrmap__ISR_INTRR_83853da9 Reg_ISR;
    rand Addrmap__IER Reg_IER;


    function new(string name = "Addrmap");
      super.new(name);
    endfunction : new


    virtual function void build();
      this.default_map   =   create_map("reg_map",   0,   4,
UVM_NO_ENDIAN);
      this.Reg_ISR = new("Reg_ISR");
      this.Reg_ISR.configure(this);


      this.Reg_ISR.build();
      this.default_map.add_reg(this.Reg_ISR, 'h0);
      this.Reg_IER = new("Reg_IER");
      this.Reg_IER.configure(this);


      this.Reg_IER.build();
      this.default_map.add_reg(this.Reg_IER, 'h4);
    endfunction : build
  endclass : Addrmap


endpackage: example_uvm_pkg
```

TABLE of CONTENT

Théophile Loubière
Simple VGA tutorial with Chisel
click

Simple VGA tutorial with Chisel

Théophile Loubière

e-mail: *theophile.loubiere@learn-fpga-easily.com*

Discussion and comments :: link

Hello FPGAmigos ! As a student, my final FPGA project was "Space Invader". We only tackled a small portion of the entire project because our instructor felt certain aspects were too complex for us, and the VGA interface was one such area (which seems utterly absurd in hindsight). My inexperienced self, freshly introduced to the mysteries of VHDL, took his word for it after merely glimpsing the daunting module filled with boilerplate code. From that moment, the VGA interface remained shrouded in an unwarranted aura of complexity in my mind. That was until I faced my GoBoard from Nandland, equipped with a VGA interface, compelling me to confront this "veteran wolf" that once intimidated my level 1 wizard self.

I decided to deviate from the standard VGA tutorial on Nandland. Instead of following it, I only referred to the VGA overview and constants. The tutorial on Nandland is presented in VHDL and Verilog, and since I'm not a fan of either, I chose to approach it with Chisel! Even if you don't have an FPGA with a VGA interface or a VGA monitor, you can still follow. I have included a cocotb simulation at the end of this article to visualize the outcomes.

In this session, I'm experimenting with a new method: I'll be using Python pseudo-code for explanations, catering to those more comfortable with software than hardware. Let me know how you find this approach.

If you're new to Chisel and eager to try it out, feel free to explore my other blog posts:

- Chisel types and operators
- Chisel Multiplexers Tutorial
- Chisel Registers Tutorial
- A little project LEDs garland with ShiftRegister

What is VGA ?

VGA, an acronym for Video Graphics Array, is a video interface standard originally launched by IBM. It quickly rose to popularity and served as the precursor to modern interfaces like DVI, HDMI, and DisplayPort. If you have an older or budget-friendly display monitor, take a look at its back. You'll likely find a blue, trapezoid-shaped connector—that's the VGA port.

How to design a VGA module ?

Designing Step by Step

As always the code is available on my Github.

Nested for loops to traverse the image

The underlying principle of VGA is straightforward. An image is essentially a two-dimensional array of pixels, each defined by RGB (Red, Green, Blue) values. The VGA protocol displays these images by printing each frame pixel by pixel, following a left-to-right and top-to-bottom order. For this example, we'll set the resolution to the standard 640x480.

In Python, this can be effectively represented using two nested for loops:

```
# pseudo code
total_column  = 640 #image width
tolal_row     = 480 #image height
for column in range(total_column):
        for row in range(tolal_row):
                send_monitor(pixel, column, row)
```

In Chisel, the concept of two nested for loops can be translated into two counters. Below is an implementation of a counter that increments when the 'enable' signal is asserted. Additionally, this counter emits a "trig" signal upon reaching the 'max_count' value:

```
class Counter(max_count: Int) extends Module{
val io = IO(new Bundle{
        val enable = Input(Bool())
        val count = Output(UInt(log2Ceil(max_count).W))
        val trig = Output(Bool())
})

val reg_count = RegInit(0.U(log2Ceil(max_count).W))
when(io.enable){
        reg_count  :=  Mux(reg_count===max_count.U,  0.U,
reg_count+1.U)
}
io.count := reg_count
io.trig  := reg_count===max_count.U
}
```

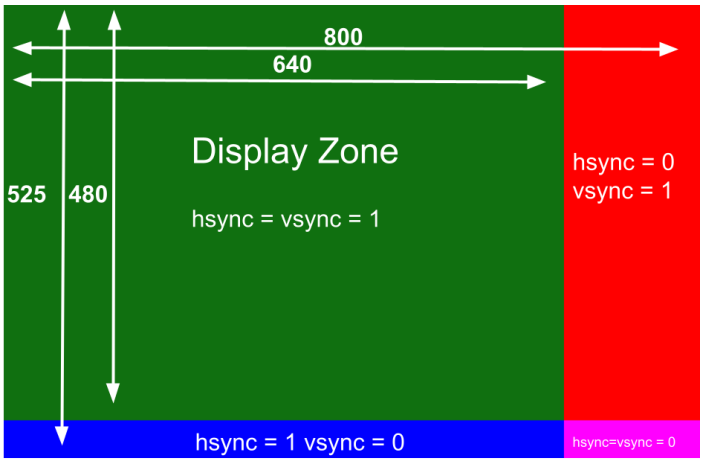
To emulate the effect of a nested loop, we require two counters, similar to the one described earlier. In this setup, the 'trig' signal from the first counter is connected to the 'enable' input of the second counter. The 'col_counter' operates as a free counter (continuously counting without stopping, hence 'enable := true.B'), while the 'row_counter' increments only when the 'col_counter' reaches its 'max_count'.

```
case class VgaConfig(
//instantiate at the end of the post ;)
TOTAL_COL : Int,
TOTAL_ROW : Int,
}
// Pseudo code for nested for loop in chisel
class NestedForLoop(config: VgaConfig) extends Module{
        // ios ...

        val   col_counter        =   Module(new   Counter
(config.TOTAL_COL))
        val   row_counter        =   Module(new   Counter
(max_count=TOTAL_ROW))
        // col_counter is a free counter
        col_counter.io.enable := true.B
        // row_counter increment only when col_counter
reach the line end
        row_counter.io.enable := col_counter.io.trig
}
```

Now that we have established the structure to traverse the frame, the next question arises: How do we synchronize our FPGA source with our VGA monitor? If a continuous stream of pixels is sent, how will the VGA monitor distinguish when a new line or a new frame begins? This is where the necessity of adding synchronization signals comes into play.

Introducing Hsync and Vsync

In VGA technology, there are two critical synchronization signals: hsync (Horizontal sync) and vsync (Vertical sync). When hsync is set to 1, it signals to the monitor that the current column is valid for pixel display. Similarly, when vsync is 1, it indicates that the current row is valid. The monitor will display a given pixel only if it falls within this valid zone, meaning both hsync and vsync need to be 1. Additionally, hsync and vsync also have intervals where they must be set to 0. This requirement leads to the concept of a "virtual frame," which is larger than the actual image we want to display. For instance, we might use a virtual frame size of 800x525.



This concept can be translated into the following pseudo-code:

```
active_column  = 640
active_row     = 480
total_col      = 800
tolal_row      = 525

for column in range(total_column):
        for row in range(total_row):
                hsync = 1 if column<active_column else 0
                vsync = 1 if row<active_row else 0

                active_zone = hsync & vsync
                if active_zone:
                        send_monitor(pixel, hsync,
vsync)
                else:
                        # when inactive, the rgb signals
should be driven low.
                        send_monitor(0 , hsync, vsync)
```

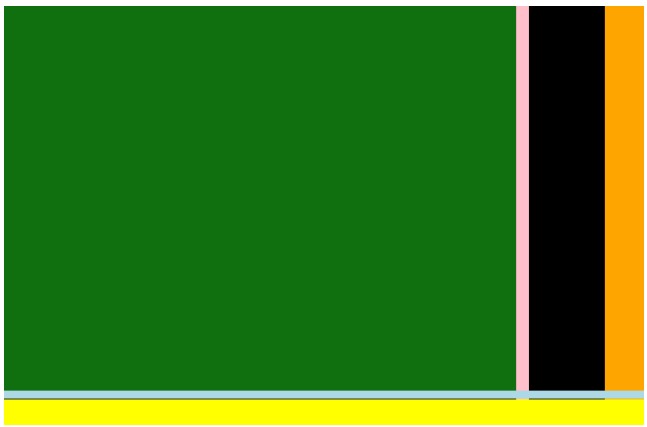
In Chisel, the implementation of this concept would be structured as follows:

```
case class VgaConfig(
        TOTAL_COL : Int,
        TOTAL_ROW : Int,
        ACTIVE_COL : Int,
```

```
        ACTIVE_ROW : Int,
}
// Pseudo code
class SyncPulse(config: VgaConfig) extends Module{
        val io = IO(new Bundle{
                val hsync = Output(Bool())
                val vsync = Output(Bool())
                val active_zone = Output(Bool())
        })

        // NESTED FOR LOOP HERE
        // ...

        // HSYNC & VSYNC
        val hsync = Wire(Bool())
        val vsync = Wire(Bool())
        // Just add two comparator
        hsync := col_counter.io.count <=
config.ACTIVE_COL.U
        vsync := row_counter.io.count <=
config.ACTIVE_ROW.U
        // Driving output
        io.hsync       := hsync
        io.vsync       := vsync
        io.active_zone := hsync & vsync
}
```

Indeed, the process essentially involves adding two comparators to generate the hsync and vsync signals. Hsync is set to 1 when the column counter is less than 640, applying the same logic for vsync and ACTIVE_ROW.

Now, you might think our VGA monitor is synchronized, but unfortunately, it's not that simple. Working with hardware often involves additional complexities. While I'm not fully versed in the specifics (I believe it relates to allowing the electron beam in cathode ray tube monitors to return to the start of the screen), our monitors require a buffer period after the end of each column and row, during which hsync and vsync remain active. Additionally, the monitor needs hsync and vsync to be asserted briefly before the start of a new line or column. These periods are known as the front porch and back porch, and they differ for hsync and vsync.



Green represents the active zone, and each different color indicates a specific type of porch: pink for the front porch horizontal, orange for the back porch horizontal, light blue for the front porch vertical, and yellow for the back porch vertical. Black areas indicate times when one or both sync signals are actually set to 0.

In pseudo-code, this concept translates to adding four comparators. These comparators check whether the column and row counters are outside the ranges specified for these porches.

```
active_column  = 640
active_row     = 480
total_col      = 800
tolal_row      = 525
FPH = 16 # Front Porch Hsync
FPV = 10 # Front Porch Vsync
BPH = 48 # Back Porch Hsync
BPV = 33 # Bach Porch Vsync

for column in range(total_column):
       for row in range(total_row):
               hsync = 1 if column<active_column else 0
               vsync = 1 if row<active_row else 0
               hsync_porch = 0 if
active_column+FPH<=column<=total_column-BPH else 1
               vsync_porch = 0 if
active_row+FPV<=row<=total_column-BPV else 1

               active_zone = hsync & vsync
               if active_zone:
                       send_monitor(pixel, hsync_porch,
vsync_porch)
               else
                       send_monitor(0    , hsync_porch,
vsync_porch)
```

In Chisel, the implementation of this concept would be structured as follows:

```
case class VgaConfig(
TOTAL_COL : Int,
TOTAL_ROW : Int,
ACTIVE_COL : Int,
ACTIVE_ROW : Int,
FPH : Int,
FPV : Int,
BPH : Int,
BPV : Int
)

//the counter here ...

class SyncPulse(config: VgaConfig) extends Module{
  // ios and add hsync and vsync porch
  val io = IO(new Bundle{
    // ... same as before
    val vsync_porch = Output(Bool())
    val active_zone = Output(Bool())
  })

  // NESTED FOR LOOP...

  // HSYNC and VSYNC...

  // HSYNC_PORCH and VSYNC_PORCHES
```

```
  val hsync_porch = Wire(Bool())
    val vsync_porch = Wire(Bool())

  // hsync with porch
  hsync_porch      :=      col_counter.io.count      <=
config.ACTIVE_COL.U       +       config.FPH.U       ||
col_counter.io.count     >=     config.TOTAL_COL.U     -
config.BPH.U
  // vsync with porch
  vsync_porch      :=      row_counter.io.count      <=
config.ACTIVE_ROW.U       +       config.FPV.U       ||
row_counter.io.count >= config.TOTAL_ROW - config.BPV.U

  // Drive outputs
  // ... other outputs +
  io.hsync_porch := hsync_porch
  io.vsync_porch := vsync_porch
```

And now it should work! That's really all it takes: two counters and six comparators, and you have a functional VGA setup. The next step is to encapsulate this configuration in a module and use it to display an image—in our example, a simple green screen.

```
class VGABundle extends Bundle{
val red   = Output(UInt(3.W))
val green = Output(UInt(3.W))
val blue  = Output(UInt(3.W))
val hsync = Output(Bool())
val vsync = Output(Bool())
}

class VGA(config: VgaConfig) extends Module{
        val io = IO(new VGABundle)
        val sync_pulse = Module(new SyncPulse(config))
        // rgb signals should be driven low outside the
active_zone hence the mux
        io.red   := Mux(sync_pulse.io.active_zone, 1.U,
0.U)
        io.green := Mux(sync_pulse.io.active_zone, 7.U,
0.U)
        io.blue  := Mux(sync_pulse.io.active_zone, 1.U,
0.U)
        io.hsync := sync_pulse.io.hsync_porch
        io.vsync := sync_pulse.io.vsync_porch
}

// The Top only to set the clock domain
class Top(config: VgaConfig) extends RawModule {
        val clock  = IO(Input(Clock()))
        val vga_io = IO(new VGABundle)
    // no reset on GoBoard
        withClockAndReset(clock, false.B){
                val vga = Module(new VGA(config))
                vga_io  <> vga.io
                }
}

// The main that generate the verilog
object Main extends App{
        val vga_config = VgaConfig(
                TOTAL_COL  = 800,
                TOTAL_ROW  = 525,
                ACTIVE_COL = 640,
                ACTIVE_ROW = 480,
                FPH        = 16,
                FPV        = 10,
                BPH        = 48,
                BPV        = 33
        )
        (new chisel3.stage.ChiselStage).emitVerilog(new
Top(vga_config),     Array("--target-dir",     "build/
artifacts/netlist/"))
}
```

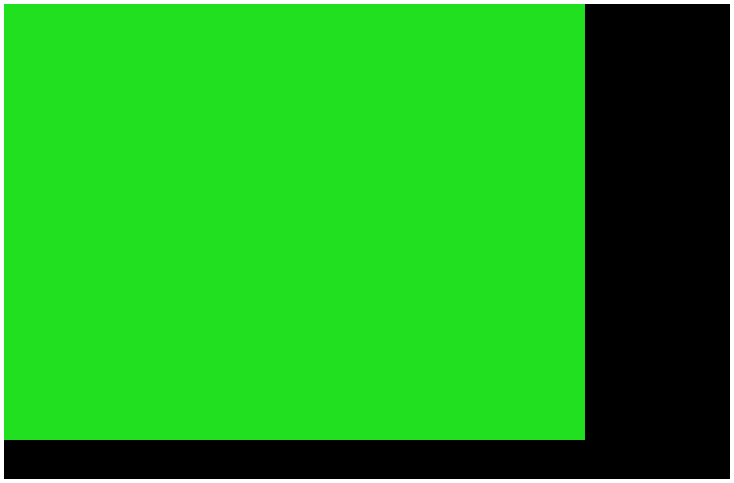
Generate a frame with Cocotb

If you don't have a GoBoard or any VGA connector, cable, or similar equipment, and yet you're still eager to witness the marvel of a green screen, don't worry! Cocotb and Verilator are here to help.

Originally, I planned to write a testbench in case of any bugs in my design. However, as I vanquished the "veteran wolf" of VGA complexity now that I am a wizard level ?? (you tell me haha), I didn't encounter any issues to troubleshoot... . Nonetheless, this experience serves as a foundation for my next tutorial (perhaps a Conway's Game of Life) and as a tool for you to verify if your design works as intended.

```
# Cocotb import
import cocotb
from cocotb.triggers import RisingEdge
from cocotb.clock import Clock
# Import to create image
from PIL import Image
import numpy as np

# Constants
TOTAL_COL  = 800
TOTAL_ROW  = 525
ACTIVE_COL = 640
ACTIVE_ROW = 480
FPH = 16 # Front Porch Hsync
FPV = 10 # Front Porch Vsync
BPH = 48 # Back Porch Hsync
BPV = 33 # Bach Porch Vsync

@cocotb.test()
async def dump_frame(dut):
        # Declare and start the simulation Clock
        clock = Clock(dut.clock, 40, units="ns")
```

```
        cocotb.start_soon(clock.start())

        # Variable declaration
        num_cycles = 800*525 # one frame
        rgb_frame  = []
        pixel_line = []
        col        = 0
        row        = 0

        # For loop : every pixel of a frame
        for cycle in range(num_cycles):
                await RisingEdge(dut.clock)
                # Get value from dut
                hsync = int(dut.vga_io_hsync)
                vsync = int(dut.vga_io_vsync)
                red   = int(dut.vga_io_red)
                green = int(dut.vga_io_green)
                blue  = int(dut.vga_io_blue)

                # Pixel in pillow a Uint8 whereas they
are 3 bits in my design.
                # We need a shift to see something
                pixel = [red<<4,green<<4,blue<<4]

                col+=1
                pixel_line.append(pixel)
                if col==801:
                        rgb_frame.append(pixel_line)
                        pixel_line = []
                        col=0
                        row+=1

        # Creating the frame with pillow and numpy
        frame     = np.array(rgb_frame, dtype=np.uint8)
        new_image = Image.fromarray(rgb_frame)
        new_image.save("vga_frame.png")
```



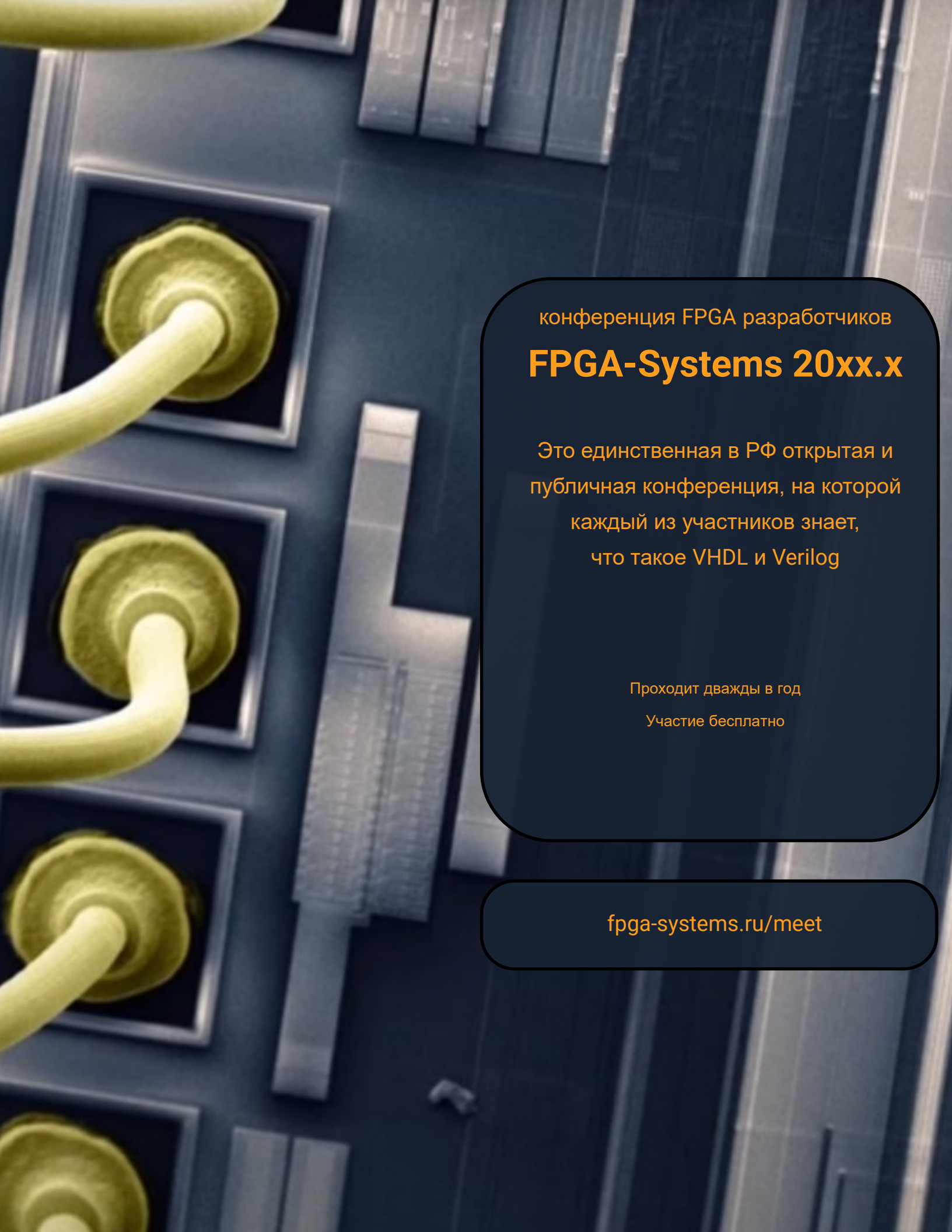
Here ! An amazing green screen !

Conclusion

And there you have it, fellow FPGA adventurers! We've journeyed through the realm of VGA, transforming a seemingly complex topic into an understandable and approachable project. From counters to comparators, and finally to the grand unveiling of a green screen, our expedition into the world of Chisel and VGA has been both enlightening and rewarding.

Remember, the world of FPGA isn't just about conquering technical mountains—it's also about the thrill of discovery and the satisfaction of solving puzzles that once seemed daunting. As we wrap up this tutorial, I hope you feel empowered to tackle your own FPGA projects, no matter how challenging they may seem at first glance.

Stay curious, keep experimenting, and don't forget to share your FPGA conquests with the community. Whether you're a novice wizard or a seasoned sorcerer in the FPGA realm, there's always a new adventure waiting around the corner. Until next time, keep exploring and happy coding!



конференция FPGA 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

Участие бесплатно

fpga-systems.ru/meet

Поддержи выход следующего номера
boosty
yoomoney

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 :: № ALFA (state_0)
Первый журнал о программируемой логике